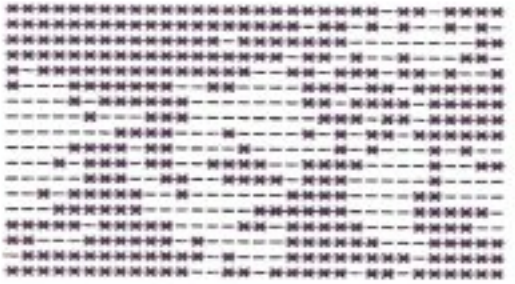
COMAL TODAY 17

page 70

COMAL Today
6041 Monona Dr
Madison, WI 53716



WHAT IS IT? (see page 30)

Bulk Rate
U.S.Postage
Paid
Madison, WI
Permit 2981

If your label says
Last Issue: 17
You must renew now
Use order form inside.

COMAL Today Issue 17 July 14, 1987

Publisher
COMAL Users Group, U.S.A., Limited
6041 Monona Drive
Madison, WI 53716

Editor
Len Lindsay

Assistants
Richard Bain
Maria Lindsay
David Stidolph
Geoffrey Turney

Artwork
Bruce Brown

Contributors
Richard Bain
Marty Balash
Jack Baldrige
Captain COMAL
Glen Colbert
Linda D
James Elrich
Ora L. Flaningam
Peter Griesbauer
Eric Haas
Gerald Hobart
Bob Hoerter
Chris Johansen
Dick Klingens
Len Lindsay

Contributors
Bob McCauley
Mel Moore
Bill Nissley
Charl Phillips
D Bruce Powell
Joel Rea
Sid Seiferlein
Mark Skopinsky
David Stidolph
Sr. Anne Stremlau
Telenova
Jim Ventola
David Vosh
RobertW32

Editors Disk

There are some major COMAL happenings to announce this issue. On the next page, Richard Bain tells of the arrival of many new COMAL implementations. I am happy that COMAL remains standardized and will now be available on several popular computers. With the arrival of these new COMALs, you need access to the COMAL Kernal standard. See page 40.

And in addition to the COMAL implementations being compatible with each other, we now have compatible books! Yes, the **Doc Box** has arrived. Page 16 explains our considerations when choosing this publishing standard.

The **Doc Box** is a mini-binder/box that can hold about 500 pages. Since many of our books are only 50-100 pages, one **Doc Box** will hold several COMAL books, all in the same place, easy to find and well protected (no curled corners or lost books). The cover of each book will come on heavy cardstock slightly wider than the regular book pages. The name of the book will be printed along the side edge of the cover; a perfect tab sheet!

Our **Doc Box** binders are quality D-ring, cloth covered, the same as IBM uses, including a matching slipcase! UniComal's IBM PC COMAL is already distributed in a Doc Box. And when released, the new CP/M and Mytech IBM PC COMAL systems will come packaged in a **Doc Box**. Place them side by side on a book shelf and the tops of the boxes provide a flat even surface (we use the tops as another shelf).

To celebrate this new standard, until June 30 we'll give you a **free** **Doc Box** if you order $20 or more of our new **Doc Box** style books. Or buy one **Doc Box** style book and get a **Doc Box** for just $5. Limit one special per subscriber. Use our special insert card.

If you looked at our redesigned order form on the last page, you will have noticed one even better aspect of the **Doc Box**: the books are much cheaper when we don't have to bind them! You save about $5 per book! You can use that $5 savings to buy the Doc Box.

You will also notice a few new books. With a standard book format, we can finalize some manuscripts that have been in the works for up to a year. See page 17.

We also are releasing 12 European COMAL 2.0 disks. Now you can see the programs being shared overseas. Three disks are from England, and 9 from Holland. Normal disk prices apply, but as an introduction special, get the full set of 12 for only $24.95 plus $2 shipping.

Our latest disk is the Data Base disk, available for both 0.14 and 2.0. It has several COMAL data base systems, along with sample data files. You should find it interesting.

I saw an interesting cable at the Commodore Show sold by Micro R&D (3333 So. Wadsworth A-104, Lakewood, CO 80227) for $19.90. It adapts a C64 power supply plug into the **unique** square type plug the C128 requires. This allows you to run your C128 from a C64 power supply.

With the arrival of a truckload of **Doc Boxes** we must clear out our inventory of backissues of *COMAL Today* (issues 6-15). We are willing to let them go **by the boxful** for **free** to schools or groups, if they pay the $50 per box estimated UPS shipping costs. There are about 100-150 copies per box (you can mix issues in a box). Along with those issues, you'll need extra **Indexes** too! We'll ship you Indexes for just $99 per box (150 copies), UPS shipping included. (USA only - Canada is higher). All others take note as well. Complete your collection of *COMAL Today* while you can. Only $1 per issue.

We are printing QLink messages again this issue since they were a favorite in past issues. ■

COMALites Unite

by Richard Bain

I have been very busy these last two months testing new versions of COMAL. We received 2.0 versions for CP/M, the Commodore 128 in 128 mode, two new versions for the IBM (by UniCOMAL and Mytech), and a version for the MacIntosh. Apple COMAL is steadily progressing, but is not yet ready for testing.

I put together a disk of programs for the new COMAL systems, including short routines such as a **type** procedure and a **file'exists** function. Longer programs were taken from *Today Disks* and *Cartridge Demo Disks*. These included the *Star Trek / Doctor Who database* programs, a calendar printing program, the game of *Nim*, and the *Tower of Hanoi* to name a few.

I soon realized that I was testing more than just new versions of COMAL. I was also testing one of the basic principles of COMAL: compatibility. How many of you have tested your COMAL programs on any computer other than Commodore? By translating a dozen programs or so, I quickly appreciated how standard COMAL is. The new versions of COMAL are reviewed separately in this issue. Now, I want to talk about compatibility.

Although no two versions of COMAL are 100% compatible, programmers **can** write programs to work unchanged in all versions of COMAL 2.0 (some changes will be necessary for COMAL 0.14). I can't give a complete list of what to do to make your programs work in all versions of COMAL, but I can give a list items to avoid if compatibility is important to you.

1) Don't depend on Commodore ASCII. Don't use **CHR$(147)** for **PAGE**, don't use the graphics character set, and don't assume that **CHR$(65)** is a lower case **a** (in CP/M and IBM COMAL it isn't).

2) Don't have a **CLOSED** procedure or function call an open one unless the open one is nested inside the **CLOSED** one. Doing this is forbidden by the Kernal and the results are different in versions of COMAL which allow it. It is generally more compatible to nest procedures than to use **IMPORT**.

3) Don't use shortcuts for substring notation. If you want the third character of string **a$**, use **a$(3:3)** instead of **a$(3)**.

4) Don't depend on a specific screen size for formatted output. Common screen types vary from 40 to 80 columns with either 24 or 25 rows. Windowing on the newer machines adds even more variety.

5) Don't depend heavily on packages. Most are machine specific.

6) Device names such as **lp:** for a printer and **0:** for a disk drive are not standard, but it is easy to use the **CHANGE** command to quickly convert them as needed.

With these tips in mind, I took the original c64 COMAL 2.0 programs and changed them on the Commodore to the point where I thought they would run under CP/M, but knew they still ran on the Commodore. Then I use the *Big Blue Reader CP/M* to translate them to CP/M disk format and make any final changes. Next, I transferred them to UniComal IBM PC COMAL. Finally, I entered them into Mytech COMAL. As a general rule, I had to change about 1 line in 50 to convert a program from one version to another. (Most changes were minor.)

I had a lot of fun learning about the new versions of COMAL and should be able to help people who want to move their old COMAL programs to the new versions. When the new versions are released, the COMAL Users Group will have demo/starter disks ready. ■

Bug Fixes

TRANSPORT FILES

Several people have asked us how we convert articles and listed program files to and from Commodore, IBM, and CP/M formats. We use a COMAL program to change word processor format codes, but the hard part is converting files from one disk format to another. We tried several programs on the c128 with the 1571 to convert file formats. The *Big Blue Reader* is by far the best. We use the ASCII translation option when converting COMAL programs so they are ready to enter directly into another computer. We avoid the ASCII conversion when translating word processor files to keep the format codes intact. We can copy several files at once by choosing files from the disk directory before the copy process begins. ■

SIDEWAYS PRINTING

COMAL Today #12 has two programs for printing spreadsheets sideways; one for 60 row sheets and one for 80 row sheets. *Sideways80* works fine, but not *sideways60*. The problem seems to be in the ASCII translations. I came up with a simple fix. Add the **bold** line to **proc print'one'char**:

```
PROC print'one'char
  p:=line$(pt'chr#)(pt'ln#) IN ascii'str$
  IF p=0 then p:=ORD(line$(pt'chr#)(pt'ln#))-31
  PRINT FILE 3: table$(p-1)
ENDPROC print'one'char
```

The problem was that capital letters in the print file were not being found in **ascii'str$**. So, if the character is not found, I take the **ORD** value and translate it from the lower case range to the upper case range.- D Bruce Powell

[The problem noted in this program is that it failed to correctly translate the Commodore ASCII codes to true ASCII. Commodore uses two ORD values for capital letters. One range is from 97 to 122 for A to Z. The other more common range is from 193 to 218. Sideways60 only translated the second set of values. The fix mentioned above is all that is needed for the first set. Note: the sideways80 program uses COMAL's built in ASCII translation feature.]

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

SMON AND SUPER CHIP

James Elrich of El Dorado, KS notes that the *Smon* monitor described in the *COMAL 2.0 Packages* book does not work with Super Chip. The problem is that Super Chip's Autostart package interferes with Smon. However, the Cmon package on Today Disk #10 works fine with Super Chip. ■

Fourth Annual
COMMODORE
Computer Show

Len Lindsay will be at the show to unveil several new COMAL 2.0 implementations, answer your questions, and present two or three special COMAL sessions. Watch for the COMAL license plate marking the COMAL booth.

The first MARCA show (in 1984) was the first all Commodore show in the country to be produced entirely by users, and has since sparked a number of similar shows around the country. The show acts as a showcase not only for new products and manufactures, but for proven favorites as well (COMAL was at the first MARCA show too).

DATE/TIME: June 20-21, 10am-6pm each day.

SPEAKERS: Len Lindsay, Jim Butterfield, Dick Immers, Lou Sanders and others.

ACCOMMODATIONS: Betsy Ross Inn (609) 665-7740, special MARCA rates: $38.75/$45.95

LOCATION: Betsy Ross Inn & Convention Center, 5 minutes from downtown Philadelphia.

For more information, please contact:

M.A.R.C.A., c/o Martha Young
P.O. Box 1902
Martinsburg, W. VA 25401
(304) 263-8264

Ninth year
The TRANSACTOR
Tech/News Journal for Commodore Computers

This special note is being prepared last minute by the *COMAL Today* staff. In order to provide a special back cover reference chart for you, we moved the Transactor Ad to page 5. Now, we find out that the ad can't easily be printed on a web press (which prints the inside 80 pages). With only half a day to correct the situation, we had to redo our master for this page. We can't come up with an ad as classy as the usual Transactor ad, but we can provide you with some special news. The Transactor is moving into a brand new large office!

Congratulations Karl, Richard, Chris, and Nick!

The Transactor has been a superb publication for the past eight years. Not many other computer magazines have stuck it out that long. If you are a serious Commodore computer user, you should benefit from a subscription to the Transactor. Their current 80 page issue boasts a circulation of 72,000. A 6 issue subscription is only $15. Order from their new address:

The Transactor
501 Alden Road
PO Box 3250
Markham Industrial Park
Markham, Ontario L3R 9Z9
Canada
(416) 737-2786

Watch for the new Transactor Ad next issue. *By the way, did you notice that the Transactor mascot, Duke (a dog), is hidden somewhere on each of the Transactor covers? Calvin the COMAL Turtle is not that lucky (though he did sneak into page 4).*

This page of last minute news and information was brought to you by:

The Transactor

QLink Message Base

Doctorwho.db - RobertW32 04/05/87

I purchased the AHOY! program disks for the last two months with COMAL on the reverse side. There are database programs on them which I cannot get to work. Can you tell me why **doctorwho.db & star'trek.db** are not able to find the files **ran.drwho & ran.startrek** respectively? Thanks for any help you can give.

Duplication problem - ComalToday 04/05/87

We have random access files on the disks we sent to AHOY. They worked fine on our copy sent to them, but were ruined on duplication.

There is no way to fix the corrupted files. You may want to wait for AHOY to correct the problem. Or you can send us $2 and we will send you the **correct** files on a disk in a box mailer. Send request to:

COMAL Users Group USA, AHOY Disk Fix
6041 Monona Drive, Madison, WI 53716

COMAL Brochures? - MELM 03/29/87

Is there a brochure that I could get 10-20 copies of so I could mail them out or otherwise give them away to people around here who are interested in COMAL? Thanks ... Mel Moore

Send ... - Captain C 03/31/87

Just EMail me your name / address and ask for the 24 page COMAL INFO booklet (along with how many copies you need). We will send them to you at no charge. Thanks ... Len Lindsay

Koalasaver - Charl P 04/05/87

Wanted: a COMAL 2.0 procedure to save a multicolor graphics screen in **koalapainter** format. Failing that, does anyone know the exact locations of screen memory, color memory, etc. so I can write my own? I've gone through the Index, and am about to go through the succeeding issues of CT, but would hate to re-invent someone else's wheel.

I'm working on a program to allow me to trace a paper sketch/image by putting it on the Koala Pad, and "drawing" over it. This doesn't work in KoalaPaint because the active surface of the pad is not the same proportion as the screen, rendering the tracings distorted horizontally.

The entire program was written in one hour, using the built-in graphics commands of the 2.0 cartridge, all except the save procedure. I'm currently using SAVESCREEN, but it doesn't create files that I can load into KoalaPaint.

I LOVE COMAL 2.0!!!!!! Charl P.

Graphics Memory - ComalToday 04/05/87

A picture is broken up into 3 distinct areas:
Hi-Res Color memory (1,000 bytes)
Multi-Color memory (1,000 bytes)
Image Bitmap (8,000 bytes)

Both color memory areas are located in the same address space ($d800-$dbe7), but in different BANKS. The Hi-Res color memory is stored in the lowest RAM bank available when the I/O block and the Character ROM is banked out. The Multi-Color memory is shared with the text screen color memory, so some bleed through is likely. The BITMAP is stored at $E000 and takes up 8,000 bytes.

I hope this helps. Perhaps the 2.0 program "koala'to'2.0" on cart demo disk #3 might help.

How about the other way? - Charl P 04/08/87

Is the "koala'to'2.0" program ML or COMAL? If it's COMAL, then it shouldn't be too hard to reverse, right?

"koala'to'2.0" - COMALite D 04/09/87

The program is 100% COMAL. I don't think it even uses packages, except for the title screen.

The program translates a Koala SAVED picture into a COMAL SAVED picture. It would not be difficult to convert this program to go the other direction. Hopes this helps.

more»

PROC simons - EricH10 02/11/87

If anyone is interested, I have found that the mysterious simons command in the superchip does the following:

1) Disables run/stop-restore
2) Stores the integer parameter in $a1 & $a2
3) Exclusive ORs $a1 with $a2 and stores the result in $a1 and $66.
4) Generates a floating point random number.
5) Moves the contents of $a1-$a5 to the address pointed to by $63
6) Sets the stack pointer equal to the contents of $a2
7) Jumps to $00a0

Locations $a0-$a2 are the software jiffy clock so, unless interrupts are disabled, the system is constantly changing these locations. It's no wonder it causes a system crash every time its called.

proc simons - Icarus 02/11/87

Hmmm. Any idea why it does all that?

A WINNER !!! - Captain C 02/12/87

I have to check with the masterminds behind the Super Chip, but it sounds like Eric got it!

Why all that, Icarus? Well, its a long story, really an inside joke! Want to hear it?

Oh, Eric: if you are right (verification pending), you will get a COMAL prize. EMail me about it in a couple days, or we can do it at the meeting here on QLink tonight.

Beginner question - Marty5 02/22/87

I just downloaded COMAL 0.14 and I'm writing my first program. Everything was going fine until I got a "stack overflow" message. The program is about 140 lines. I need 2 arrays, and I'm using:

```
dim location(21,40)
dim location'status(21,40)
```

It points to one of these lines when I get the error message. Since I have no documentation other than some program listings, any help would be appreciated. Marty5

SIZE - Icarus 02/22/87

One thing occurs to me, Marty. After your COMAL system is loaded, type SIZE and hit <return>. You will see something like 11,000 Bytes Free if you have the **expand'ram** proc in your "HI" program (chained by the BOOT). If you see about 9,000 Bytes free, it means you have the fundamental COMAL 0.14. For the last two years, most of us run a HI program that finds the other free bytes, so that our disk loaded COMAL systems have more RAM. Even so, you have to be careful not to DIM too many arrays. Another cause of stack overflow, is using too much recursion. I'm sure you'll get more definitive answers tacked on to this one shortly. Welcome to COMAL. I know you will enjoy it. Regards, Jim Ventola

Beginner question - Marty5 02/23/87

Thanks for the response, Jim. After I logged off qlink, I re-read one of the few COMAL magazine articles I could find and found something that worked:

```
list "progname.l"
new
enter "progname.l"
save "progname"
```

According the article, this procedure rebuilds the name table but all I know is the program went from about 12 to about 8 blocks in size, and it ran fine. Can anyone explain?

Name tables - COMALite J 02/24/87

All variables, proc/func names, labels, etc. are stored in what is known as the "Name Table". This means that repeatedly using a long name like "wait'for'spacebar" doesn't take up all that room each time -- only once, and then only in the name table itself -- not the actual program!

more»

This "name table" holds the length of the name-table entry and the name itself. Each name entry has a corresponding entry in a "name value table", which holds the type of name (real var, integer var, string var, real array, int array, str array, label, real FUNC, int FUNC, PROC or undefined), and a pointer to its value.

Each time you enter a new name into your program, even accidentally (ie, typing LOST instead of LIST), COMAL builds a new entry for it. These entries remain undefined, but get SAVEd and LOADed with your program. The only way to clear them out is to re-type the program! Fortunately, COMAL can do this for you. When you LIST to disk instead of saving, you save the text (source) of your COMAL program to disk as a SEQ file! When you ENTER that file, it is the same to COMAL as if you typed it back in. COMAL starts with empty tables, and builds new ones containing only those names currently in your program!

Another question - Marty5 02/24/87

Are there any general rules for more efficient programming in COMAL? My program has run out of room. I don't think I've declared too many arrays, but a few of my procs are kind of long. Would breaking down a long proc help any? Also is there a **find** or **replace** command in 0.14? Thanks, Marty

More efficient programs. - EricH10 02/27/87

Marty, try making your arrays into integer arrays, instead of real arrays. An integer array only uses two bytes per element, while a real array uses five. Also, if you have any CLOSED procedures or functions, try to make them open. A CLOSED procedure or function needs more stack space than an open one. And avoid recursion. Recursion uses lots of stack space.

Thanks - Marty5 02/27/87

Thanks for your help! I'm uploading the program in question - just one week after I downloaded COMAL 0.14. I think I'll get the book, now... Marty [*direction'game, is on the 0.14 side of Today Disk #17*]

ESC from INPUT? - FROM: Icarus 02/11/87

Is there a way to detect the STOP key being pressed when COMAL is waiting for user input? I would prefer that method to having the user input 0 to Quit the loop, but I haven't been able to implement it. I would like this for both versions, if possible. Thanks. //Jim

PS. In COMAL 0.14, if there is an input error, the error message is printed on the next line, scrolling the screen if the input is requested near the bottom, where I want it. Any way to prevent it?

No, and no --- or maybe? - Captain C 02/12/87

Well, as it stands now, the answer to both your requests is no. You can't detect the STOP key while in an INPUT (though this would be nice). You can't stop the input error message in 0.14 ... so avoid line 25 if that matters to you.

In 2.0, without using TRAP ... HANDLER the same thing applies. You can avoid this by doing your own error checking with 2.0 though! For example:

```
LOOP
  TRAP
    INPUT "Enter a number: ":num;
    PRINT "Thank you."//on same line
    EXIT // exit TRAP and LOOP
  HANDLER
    PRINT "<-- Oops!"//still same line
  ENDTRAP
ENDLOOP
```

With the ; after the input statement, the cursor stays on same line as input ... so you don't get the scrolling. My OOPS message is not the best way to handle it (if you type "a" for the reply), but it does show how the scrolling is avoided. Clear? Regards ... Len

more»

Trap input - LindaD 02/12/87

If you have the Comal Handbook check out the area under TRAP - I think it will allow you to check for pressing the STOP key, though I'm not sure if it will work during INPUT.

TRAP - HANDLER (see book) could be used to fix your screen scrolling on an INPUT error. This function allows you to put in a PROC (or any other statement) to perform if your program encounters an error at the particular area where it placed. In your case you would place it after the input prompt and if an error was encountered you could have it print the error message elsewhere besides the line after the INPUT prompt and then return to ask for INPUT again. If you need more info and/or examples (and don't have the book) let me know and I'll try to leave specific examples. I haven't tried either of these things in combination with INPUT so I don't guarantee how they will work - I have used them successfully in other applications. TRAP - HANDLER is very useful in file applications where you don't want your program to bomb out in the middle of reading or writing to a file. Hope this helps, Linda
P.S. I'm not sure if these features are implemented in 0.14.

Mytech COMAL does! - AppleCOMAL 02/12/87

Your basic question is detecting the stop key during an INPUT. This cannot be done in UniComal COMAL. However, Mytech COMAL has a TRAPESC(<num>). If you use 0 for <num> then the STOP or ESC key is not trapped and will cause the program to stop. If you use 1 for <num> then the STOP or ESC key is trapped and is ignored (except for the ESC function). If you use 2 for <num>, the ESC or STOP key will generate an error that can be trapped.

I really like this method. In Apple COMAL, I would like to combine the TRAP ESC+ and TRAP ESC- and use another symbol after ESC that would indicate that an error should be generated if the STOP key is pressed.

Trap--Thanks - Icarus 02/12/87

It's a shame that UniComal didn't realize that programmers might want to have the user hit the STOP key to signal his or her desires. I have to make them input a zero to tell the program that they have no new number to input. Oh well.
Linda, thanks for the tips. 0.14 doesn't have TRAP...HANDLER. What I did in the 2.0 version was to use recursion. After HANDLER, I just re-called the whole PROC, since in this program there is only one datum active at any time. Thus, to the user the error is invisible. The program just waits patiently and without complaint until it gets a number. Of course, if that number is too big, I have coded in messages about that. Best, jv

STOP "within" INPUT - Xojo 03/10/87

You can go halfway, if you're willing for the user to understand that s/he must hit <stop>, then <return> in 2.0, or *hold*down* <stop>, then hit <return> in 0.14 (see the bible, "The COMAL Handbook", under TRAP, Note 2). In either 0.14 or 2.0, place a TRAP ESC - before the input statement. Style and safety demands you have a separate variable, used as a boolean flag, in either version, assigned following the input statement -- say, "GiveUp", thus: GiveUp := ESC. Later on in your program you can re-enable the <stop> key with TRAP ESC +, then look to see if GiveUp is true before you look at the input string.

This is the best I have come up with for aborting input. I understand that the <stop> key generates a hardware interrupt. Some firmware hacker should be able to write a patch to COMAL's and the KERNAL's interrupt handler to both set the value of ESC and do an EXIT or **return** to the next higher calling process. I haven't got into the nuts and bolts of it, yet. ■

COMAL Clinic

SCOPE RULES

Scope rules govern how variables are handled when a **CLOSED** procedure/function calls a open one. This is forbidden by the COMAL Kernal, but allowed by some COMALs.

When a **CLOSED** procedure is executed, only the names created within that procedure are known. The program below sets the variable **static** to the value **TRUE** (1). When the **CLOSED** procedure **closed'proc** is called, all the names known to the main program are ignored. They are considered *out of scope*. The procedure must **IMPORT** (bring the old definition into scope) the procedure **open'proc** to call it. It assigns a **new** variable **static** (remember this is a **CLOSED** procedure which can't know about or change any variables from the main program) the value of **FALSE** (0) and calls **open'proc**.

This is where static or dynamic scope rules differ. Under dynamic scope rules, the variable **static** comes from the **CLOSED** procedure which called the open procedure, and has a value of 0 (**FALSE**). Under static scope rules the variable **static** comes from the global definition because the procedure is defined as being open, and has a value of 1 (**TRUE**) from the main program.

```
static:=TRUE
closed'proc
//
PROC closed'proc CLOSED
 IMPORT open'proc
 static:=FALSE
 open'proc
ENDPROC closed'proc
//
PROC open'proc
 IF static THEN
  PRINT "This COMAL has static scope rules"
 ELSE
  PRINT "This COMAL has dynamic scope rules"
 ENDIF
ENDPROC open'proc
```

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

DIV and MOD

Recently, while working on programs for doing high precision arithmetic, I finished a COMAL 2.0 version, then went to work on a 0.14 version. The results were discouraging.

COMAL 2.0 and 0.14 differ in the way they perform **MOD** and **DIV** operations. When using these operators on a negative number, COMAL 0.14 gives the so-called true modulus, which is always positive, while a negative number in COMAL 2.0 gives a negative **MOD**. For example, the expression **(-55) MOD 10** gives -5 in COMAL 2.0 and 5 in COMAL 0.14. Similarly, the value of **(-55) DIV 10** equals -5 in COMAL 2.0 and -6 in COMAL 0.14. Neither is wrong, but I had quite a time until I discovered it.
Jack Baldrige - Boulder, CO

*Yes, the two versions of COMAL do handle DIV and MOD differently for negative numbers. The COMAL Handbook defines X DIV Z as INT(X/Z) and A MOD B as A-INT(A/B)*B. With a little work, you will see that this is consistent with the COMAL Kernal presented later in this issue. This is how COMAL 0.14 handles DIV and MOD.*

*COMAL 2.0 uses a different definition. X DIV Y is the same as INT(ABS(X/Y))*SGN(X*Y). X MOD Y is the same as X-(X DIV Y)*Y. Note, this definition of MOD is worded the same as in the Kernal, but since COMAL 2.0 uses a different meaning for DIV, the meaning of MOD is also changed.*

Final note: DIV and MOD are only different in COMAL 2.0 when using negative numbers. Be careful when typing examples to test them. **-55 DIV 10** *is treated as* **-(55 DIV 10)** *instead of* **(-55) DIV 10**.

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

more»

FRAME

The **FRAME** command in COMAL 0.14 defines the area that the turtle pen is allowed to draw in. A side effect of this command is to move the turtle to the center of the frame after issuing the **HOME** command.

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

ENTER 120 CHARACTERS

The Commodore screen editor imposes an 80 character per line limit on COMAL. However, C64 COMAL 0.14 & 2.0 can actually accept 120 characters per line. To do this, you must type the line in a word processor and **ENTER** it back into COMAL. This method is not recommended, but may be useful to someone desperate to include meaningful variable names in a long procedure header list. Note, the line will list correctly and can be renumbered or deleted, but you will not be able edit it (this would require entering more than 80 characters into the Commodore screen editor). We used this procedure for our new Generic Data Base system that we plan to release in *COMAL Today #18*.

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

FREE DISK SECTORS

Today Disk #17 includes a 2.0 procedure Bob McCauley sent us to quickly (2.5 sec) look at the BAM and count free sectors on the disk. It also returns the disk name and ID. If you only want the number of free sectors, use the integer function of the same name. They require the **Rabbit** package. *[The procedure and function cannot be used together in the same program unless one is renamed.]* To use them:

```
free-disk(count#,name$,id$) // procedure
// or
PRINT free-disk# // function
```

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

PRECISION

This short program can be used to determine the floating point precision of your computer. The C64 has about 9 digits. The IBM PC has about 15.

```
digs:=1
WHILE (1+digs/2)<>1 DO digs:=digs/2
PRINT USING "## significant digits": INT(
-LOG(digs)/LOG(10)) // wrap line
PRINT USING "## binary bits": -LOG(
digs)/LOG(2) // wrap line
```

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

LINE LENGTH & PRINTERS

Question: At Queen Charlotte High School, we use condensed print mode to generate a table of trigonomic functions 136 characters wide. For some reason, COMAL 0.14 will not print past the 80th position. Instead it prints a carriage return and continues printing on the next line. What can be done to correct this?

Answer: It seems that UniComal set a maximum line length of 80 in 0.14 and 132 in 2.0. When a comma is encountered in a PRINT statement, COMAL checks, and if the new position is past the maximum line length, it issues a carriage return. (Some printers overprint a line if you don't issue a carriage return at the end). The following procedure changes the maximum number of columns for COMAL 0.14. The change for 2.0 is not known at this time.

```
proc line'length(columns)
  poke 21229,columns+1
  poke 21236,columns
  poke 21340,columns
  poke 21423,columns+1
  poke 21461,columns+1
endproc line'length ■
```

How to ...

Submit Articles and Programs

Would you like to share information, programs, or articles with other COMALites? COMAL 0.14 material is especially appreciated. We will soon need programs for IBM and CP/M COMALs too. Send all submissions to:

COMAL Users Group, U.S.A., Limited
6041 Monona Drive
Madison, WI 53716

If you submit a program, please send it on disk. A printed listing of the program is not necessary. If possible, also include a text file explaining the program. **Put your name and address as remarks at the beginning of your programs.** This helps us give proper credits if they are used. Most important: label the disk with your name, address and date. Also include disk format: c64, IBM, CP/M, Apple, etc.

Articles should be submitted as standard SEQ text files on disk. If possible, also include a printout of each file on the disk. Don't include any special formatting commands in your files (we have to delete them). We use special formatting with PaperClip on our Commodore computer and Word Perfect on our IBM computer for our LaserJet printer.

Don't worry if you aren't a professional writer. Articles sent to us go through extensive editing. We actually go through over 4,000 sheets of paper while preparing one 80 page newsletter! You don't have to follow a bunch of rules, either. We rework your submissions to fit our newsletter format.

Material submitted is not returned. However, if you send us a disk, we will send one of our User Group disks back to you in exchange. Just specify which one.

Submitted material may also be used for our **READ & RUN** series disks or the *Ahoy!* disks.

Type In Programs

Line numbers are required for **your** benefit in editing a program (but are irrelevant to a **running** program). Thus line numbers often are omitted when listing a COMAL program. It is up to **YOU** to provide the line numbers. Of course, **COMAL can do it for you.** Follow these steps to enter a COMAL program:

1) Enter command: **NEW**
2) Enter command: **AUTO**
3) Type in the program
4) When done:
   COMAL 0.14: Hit «*return*» key twice
   COMAL 2.0: Hit «*stop*» key

While entering a program, use unshifted letters. If letters are capitalized in the listing it does not mean to use SHIFT with those letters. They are capitalized merely to be easy to read. The only place to use SHIFTED letters is inside quotes. Also, you don't have to type leading spaces in a line. They are listed only to emphasize structures. You **DO** have to type a space between keywords in the program.

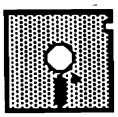
Long program lines: If a complete program line will not fit on one line, we will continue it onto the next line and add *//wrap* at the end. **You** must type it as one continuous line.

Variable names, procedure names, and function names can be a combination of:

abcdefghijklmnopqrstuvwxyz 0123456789'] [\ _

The «*left arrow*» key in the upper left corner of the keyboard is valid. COMAL 2.0 converts it into an underline. If you see an underline in a program listing, type the «*left arrow*» key. The C64 and C128 computers use a «*British pound*» symbol in place of the \ backslash. ■

Extra Programs



The following programs are on *Today Disk #17*. They are all written in COMAL 0.14.

Calculate e

Jack Baldrige wasn't content to just sit around and calculate **PI**, this high precision program calculates another familiar number.

Sample Rate Calculator

This tool by David Vosh is designed to figure out the maximum audio frequency that can be synthesized by direct digital synthesis.

Printer Routines

Several short programs are included for Epson printers. They can be used to change between pica and elite, change fonts and expand or condense text, set superscripts and subscripts, set bold and underline, turn on the near letter quality mode, and more.

Marquee

Sid Seiferlein sent us cute cartoon (he calls it a commercial). Actually, it is a two act play. The animation is impressive for COMAL 0.14.

Polynomial Fit

Ora L. Flaningam wrote a least squares polynomial fit program. The user enters the data and the program calculate the curve. It also tells you how well the curve fits the data.

Expression Analyzer

A COMAL 2.0 program is described on page 55 to analyze COMAL arithmetic expressions. We didn't want the COMAL 0.14 users to feel left out, so we dug up an expression evaluator program from the original *Sampler* disk. It doesn't give the RPN expressions, but it can do a few things that the 2.0 program can't. ■

Sililoquy

To close or not to close: that is the question
Whether 'tis nobler in the scope to suffer
The names and values of outrageous variables
Or to take up arms against a sea of tokens
And by closing end them? To de-scope: to see
No more; and by a word to say we end
The heart-ache of a thousand unnatural names
That packages are prone to, 'tis a consummation
Devoutly to be wished. To close, to purge;
To CLOSE, and then IMPORT: ay, there's the rub
For in that CLOSED state what needs may come
When we have shuffled off this open scope
Must give us pause: there's the fear
That makes calamity of a CLOSED scope
For who would bear the FILLs and SCREENs and TIMEs,
The necessary SCAN before command,
The errors of despised parameters, the LINK's delay
The verbosity of graphics and of MOS
That namespace from the user takes
When he himself might his quietus make
With a bare keyword? Who would EXTERNALs bear,
To LOAD and SAVE unto a weary file
But that the dreaded "ERROR: undefined"
The undiscovered reference form whose bourn
No value is returned, puzzles the will
And makes us rather bear the bugs we have
Than fly to others that we know not of
Thus convenience does make cowards of us all
And thus the name-space allocation
Is sickled o'er with their bale identifiers
And programs of great length and modularity
With this regard their structure turn awry
And lose their erstwhile clarity.

[This poem was the concluding page of Telenova's proposal to the COMAL standards committee. Telenova is a Sweedish company which has versions of COMAL running on European computers. You might remember them from the COMAL Standards Meeting article in COMAL Today #15.] ■

Text Reader

by David Stidolph

This program on *Today Disk #17* reads the full printer package design article (summarized in the next column). Use the cursor keys to scroll up or down through the file.

```
USE system2
DIM text$(600) OF 37
OPEN FILE 1,"txt.printer pack",READ
count:=0
WHILE NOT EOF(1) DO
  count:+1
  INPUT FILE 1: text$(count)
ENDWHILE
CLOSE
//
PAGE
top:=1
FOR x:=1 TO 24 DO PRINT " ",text$(x)
PRINT " ",text$(25),
//
TRAP ESC-
WHILE NOT ESC DO
  CASE KEY$ OF
  WHEN ""17""
    IF top+25<count THEN
      top:+1
      PRINT AT 25,1:
      PRINT AT 25,2: text$(top+25),
    ENDIF
  WHEN ""145""
    IF top>1 THEN
      top:-1
      scroll'down
      PRINT AT 1,2: text$(top)
    ENDIF
  OTHERWISE
    NULL
  ENDCASE
ENDWHILE
TRAP ESC+
PAGE
END "Done." ■
```

Printer Package

The concept behind a standardized printer package is to have a unified set of printer instructions which will allow the use of special printer commands, graphic screen dumps, and custom fonts in any program. You shouldn't need to modify COMAL code to include escape sequences for each printer. To accomplish this, printer packages should be designed for each type of printer. Each one would use the same procedure names to perform common tasks, even though the printer specific code will be different for each printer package. The user would link the appropriate package to match the printer before running the program.

Glen Colbert sent us a printer package design kit for COMAL 2.0. It is an unfinished, ongoing project that needs user support to make it work. The current version works for the Gemini printer, but the goal is to allow the same COMAL code work on other printers as well (by modifying the package only). If anyone can enhance it, **please** do so.

Several things should be kept in mind while working on a printer package. First, every effort should be made to ensure that the routines you write as a part of the package should be as functionally identical to those in the original package as is possible. Second, if the printer that you are working on doesn't support and/or can't simulate one of the functions in the package then a change of state should occur anyway. Possible exceptions to this might be a call to a graphics dump on a letter quality printer. In these exceptions, it may be wise to generate a not implemented error.

[This package, including source code and a more complete list of Glen's ideas, is on Today Disk #17. We hope that a few programmers will help create a standard COMAL package environment for the popular printers.] ■

Spin & Win

by Sister Anne Stremlau

This COMAL 2.0 game (on *Today Disk #17*) is roughly based on Wheel of Fortune. The words or phrases used come from sequential files which may be prepared by a teacher using the accompanying *spin&win'tools* program. The phrases in the lists must be no more than 15 characters long. The game is intended for teachers to use for review of such things as vocabulary, names of states, rivers, presidents, elements, etc. It is also good for the weekly spelling words.

Directions for the Game

The computer asks which list of phrases you want, gets that list form the disk, then randomly picks one of the phrases for you to guess. The wheel "*spins*" and a dollar amount which you may win is posted on the screen. Your actual winnings depend on this dollar amount and the total number of guesses required to solve the puzzle. The screen shows the number of letters and spaces and which letters you have already tried. There are five phrases per game.

Future Changes

In the future I want to rework the game to allow for longer phrases and for hints to go with each phrase. This would make it a more useful tool for schools.

Writing Your Own Lists

To prepare your own list of data, first **RUN** the *spin&win'writer* program to create a blank file. Then **RUN** the *spin&win'tools* program to enter the words or phrases (all characters are converted to upper case). If you make a mistake, keep entering the new phrases until you have finished. Then you will be given an option to go back and edit or delete your mistakes. ■

Best Books

Here are the charts of the best selling books for January, February and March:

January 1987

#1 - COMAL From A to Z
by Borge Christensen
#2 - COMAL Handbook
by Len Lindsay
#3 - COMAL 2.0 Packages
by Jesse Knight
#4 - Cartridge Graphics & Sound
by Captain COMAL's Friends
#5 - Cartridge Tutorial Binder
by Frank Bason & Leo Hojsholt

February 1987

#1 - COMAL From A to Z
by Borge Christensen
#2 - Introduction to Computer Programming with COMAL
by J William Leary
tie COMAL Today - The Index
by Kevin Quiggle
#3 - Graphics Primer
by Mindy Skelton
tie COMAL Handbook
by Len Lindsay
#4 - COMAL Workbook
by Gordon Shigley
#5 - Foundations in Computer Studies With COMAL
by John Kelly

March 1987

#1 - COMAL Today - The Index
by Kevin Quiggle
#2 - Packages Library #1
by David Stidolph
#3 - COMAL 2.0 Packages
by Jesse Knight
#4 - Beginning COMAL
by Borge Christensen
#5 - COMAL Workbook
by Gordon Shigley ■

Doc Box

For over ten years, COMAL has been maintaining its compatibility between different computers. We now are ready to extend this compatibility to COMAL information... the manuals and books! Introducing the **Doc Box**.

We have spent the last few months researching and investigating various methods of distributing manuals and books. Our considerations included:

Size

While a bigger page size gives more information at one time, it also is less convenient. Opened up on your desk, it requires quite a bit of empty desk space. Since free space next to most computers is at a premium, the biggest page sizes were ruled out. The page size that we found to be the best is 8 1/2 by 5 1/2 inches. This is the size of many of our COMAL books, including: *COMAL From A to Z, COMAL 2.0 Packages, Packages Library, Library of Functions and Procedures, Graphics Primer, COMAL Quick,* and *Cartridge Graphics and Sound*. You can create your own pages of this size by taking a sheet of standard size typing paper and folding it in half! This makes it a very efficient page size, since we can print two full pages on one sheet of paper.

Binding

We found that most people wanted to be able to open up a book, set it down, and have the page stay open. There are two ways to produce a book to meet this requirement: Spiral Bound and Binder style. Flat bound books (like *Beginning COMAL*) tend NOT to fit into this category. Nor do those stapled in the middle (like *COMAL From A to Z*).

Storage

Once you have a few COMAL books, you need a way to store them so that they are easily accessible and easy to locate as well. Here, a thick flat bound book (like *Beginning COMAL*) is ideal. Binder style books are also good in this regard. Our small books do not fare well though. You can place them side by side on your bookshelf, but they have no identification on their spine (some are stapled and don't even have a spine).

Expandable / Easy to Update

Some of our users wanted to be able to add notes to their books, or to expand them by including program listings at the end. Only the Binder style allows this. Likewise for updating an incorrect page. A page in the Binder can be removed and replaced with a corrected page.

Efficient

Thus far, COMAL books come in many sizes. Place them together on a shelf and you have a jagged skyline look. If all the books were the same height, placed next to each other on a shelf they would form a flat surface accross the tops. You could efficiently fit them together in a shelf of that height, or even place papers on top of them.

Keep It's Shelf Place

If you have several books on a shelf, sometimes it is awkward when you pull one of them out to read. Those next to it immediately fall over to the side. The one solution to this is the Binder that comes in a Box (called a **slip case**). If you want to read that manual, you only pull out the Binder. The slip case remains on the shelf. No other books are affected. When done reading, a spot on the shelf is waiting to store it.

The Solution

A **binder** for 8 1/2 by 5 1/2 pages, together with a matching **slip case** is the one method that meets all our criteria. Maybe that is why IBM uses the same size binder and slip case! ■

New Books

Now that we are standardizing on the **Doc Box** system, we are preparing to release a flurry of books, some in the works for over a year! They will be published as Doc Box style pages. The cover of each book will be a bit wider and heavier, acting as a tab sheet for that book, with the books name printed sideways on the tab. That makes it easy to flip between books in the same Doc Box.

The *Graph Paper* book/disk set came out in March, a month earlier than our April projection. It is now available in Doc Box pages (the first ones were saddle stitched). It includes a COMAL 2.0 program on disk for graphing functions, sort of an electronic graph paper (hence its name). The program is not listable, but you can copy it for archival/ backup use.

We also have reprinted Jesse's *COMAL 2.0 Packages* and David's *Packages Library #1* on Doc Box pages. We hope to have a sequel, *Packages Library #2* out soon (on Doc Box pages of course). We also are planning a reprint of *Cartridge Graphics and Sound* on Doc Box pages. Plus the *Cartridge Tutorial Binder* has pages only slightly wider than our Doc Box size. So we will only have to punch the 3 ring holes to make it Doc Box compatible.

As soon as this newsletter is done, we will begin finalizing new manuscripts. Here are the books we plan to publish by July 1987:

Packages Library #2 - a book/disk set that will include many of the packages we have released since volume #1 was published (excluding the Super Chip on Disk set). Doc Box pages only.

COMAL Collage by Frank and Melody Tymon - this book/disk set is a graphics and sprite tutorial for the C64 COMAL 2.0 cartridge. It should be about 200 Doc Box pages.

Today Tutorials - this book will be selected reprints from the first 17 issues of *COMAL Today*, updated as needed. It should apply to all COMAL implementations. Doc Box pages only.

Programming tips and notes are considered the best part of *COMAL Today* by many readers. We are now planning a collection of the best tips and notes from *COMAL Today* issues 1-17.

3 Programs in Detail by Doug Bittenger - this book/disk set explains how to construct three useful programs: Blackbook (a name and address type system), Home Accountant, and BBS.

COMAL Cross Reference by Len Lindsay - this book takes over where the *COMAL Handbook* left off. While the *Handbook* covered C64 COMAL 0.14 and 2.0, this *COMAL Cross Reference* will cover the COMAL 2.0 implementations generally available in the United States and Canada (including the new CP/M and Mytech COMALs).

Meanwhile, the following COMAL books can be ordered directly from your local bookstore:

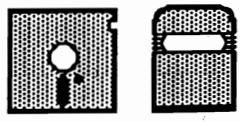
Structured Programming With COMAL by Roy Atherton, distributed by Wiley
Starting With COMAL by Ingvar Gratte, distributed by Prentice Hall
Commodore 64 Graphics With COMAL by Len Lindsay, distributed by Reston Publishing, a division of Prentice Hall (now out of print)
Adding Structure to BASIC With COMAL-80 by Max Bramer, distributed by Addison Wessley ■

CLASSIFIED ADS

For sale: Black C64 COMAL 2.0 cartridge with Super Chip installed for $75. David Eagle, 7952 West Quarto Drive, Littleton, CO 80123

For sale: Beige COMAL 2.0 cartridge, never used - $45. Also for sale, one Super Chip, never installed - $25. John Stewart, RR2, Box 511, Sierra Vista, AZ 85635, 602-378-6316 after 6pm ■

RAM Expander



The 1750 ram expander is a special storage device that you plug into the cartridge port of the C64 or C128. *[We do not have access to the 1700 or 1764 ram expanders, but expect them to work similarly.]* Normally this would not allow you to use it with another cartridge. A cartridge expander such as the *Aprospand* can be used to let more than one cartridge be plugged in at one time. The COMAL cartridge and the 1750 can be used together. All that is needed is software. It will not give you more programming room, but will give you the ability to store information at lightning speeds.

The 1750 has many potential uses. It could be used as a fast ram disk, or to store large amounts of information for processing. It could also be used to have multiple programs running and being swapped in and out of the computer.

The ram expander has 8 banks, or areas, of memory. Each bank is the same size, 65536 bytes. Making use of the expander is fairly simple. For example, to save the text screen (which starts at location 1024) to the ram expander, follow these steps (they work in both COMAL 0.14 and 2.0 and are accomplished by the TRAN procedure listed below):

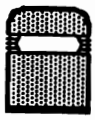
1. Put the starting address (intsa) of the text screen in registers 2 and 3.
2. Put the ram expander bank (expb) you wish the text to go in into register 6.
3. Put the starting address (expsa) where you want the text stored in the ram expander into registers 4 and 5.
4. Put the number of bytes to transfer (bytes) into registers 7 and 8.
5. Finally, tell the ram expander to start the transfer, and which direction (action) to send the data.

The following program demonstrates the technique of saving and restoring the text screen. It could also be expanded to save the color screen as well.

```
stash:=144; fetch:=145; swap:=146
PRINT CHR$(147),CHR$(14)
PRINT "This program will save the text"
PRINT "screen to the 1750 ram expander."
PRINT "Then the screen will be cleared."
PRINT "Finally, the program will restore"
PRINT "the screen from the 1750 ram"
PRINT "expander."
tran(1000,1024,0,0,stash)
wait
PRINT CHR$(147)
PRINT "The screen has been cleared"
PRINT
PRINT "Now I will restore the original"
PRINT "screen and end this demo."
wait
tran(1000,1024,0,0,fetch)
PRINT "Done."
//
PROC tran(bytes,intsa,expsa,expb,action)
CLOSED // wrap line
  re:=57088 // start of expander control
  FOR i:=re TO re+10 DO
    POKE i,0 // clear all 11 registers
  ENDFOR i
  POKE re+8,(bytes DIV 256)
  POKE re+7,(bytes MOD 256)
  POKE re+3,(intsa DIV 256)
  POKE re+2,(intsa MOD 256)
  POKE re+5,(expsa DIV 256)
  POKE re+4,(expsa MOD 256)
  POKE re+6,(expb+248)
  POKE re+1,action // start process
ENDPROC tran
//
PROC wait
  PRINT
  PRINT "Press any key to continue"
  WHILE KEY$<>CHR$(0) DO NULL
  WHILE KEY$=CHR$(0) DO NULL
ENDPROC wait
```

[Warning: this program does not seem to work reliably, possibly due to bugs in the c128 ROMs. We would like to hear from anyone who uses this or can improve on it.] ■

Video Filer System

by Bob Hoerter 

The video file system was developed at the request of my friend, Fred Sherman. He had expressed dissatisfaction with a commercial program he purchased to keep track of his video collection. I saw a opportunity to sell my beige COMAL 2.0 cartridge and do a little programing. *Videofilersystem* is on *Today Disk #17*. The main data file contains a partial listing of Fred's video collection.

Using the bird data concept from *COMAL Today #15* page 76 (i.e. all information contained in one string variable) the *videofilersystem* was created. This data base consists of a short menu program which calls four **EXTERNAL** procedures. These procedures do the following:

1) enter the data
2) create sub-files
3) edit the data
4) split long files alphabetically.

The menu program has the **Mlw'sorts** package by Robert Ross linked to it. It is significantly faster than the shell sort method used in *bird'data'base*. Note that even though the sort routines are used in an **EXTERNAL** procedure, they must be called in the main program.

The enter data procedure allows you to enter several records before the information is written to the disk. Writing each record as it is entered is too slow. The program is set to write 10 records at a time. This may be changed by the user to any number from 1 to 100.

The program uses several string functions so that only one letter is needed for a string of up to 12 letters. This saves both typing time and disk space. The entry fields are shown with any necessary prompts which pertain to that field. All data should be entered in lower case, the title will be printed in all caps, either to the screen or on hardcopy.

After entry of the record is complete, it is displayed. You are asked if it is correct and the information is stored in an array until it is written to disk. You are then prompted for another record.

The sub-file procedure lets you choose the information you wish to extract from the main data file. Prompts let you choose the portion of the record to search in for your sub-file. As the main data file is read the number of matches is printed on the screen. When the search is completed, the data is sorted alphabetically by title. The maximum number of records that may be held in the sub-file is 300. If your search produces a file with too many records you will be told to split the file into two portions sorted alphabetically by title. If the data file is less than 300 records, it is not written to disk, but the options menu lets you:

1) view the file on the screen
2) print the file
3) resort the file by other fields
4) write the file to the disk.

The edit file procedure will allow you to edit the main data file by changing or deleting records. You are prompted for a search key. As the file is read, the number of records read is printed. When a match is found, the record is printed in its raw form in separate fields on the screen. A bell sounds and you are asked to:

1) change the record
2) delete the record
3) do nothing

If you opt to change the record, then you cursor through each field. Type changes where necessary or press «*return*» to retain a field. All records changed are shown in original form and the revised form. You are asked if you want to make the change. All records marked for deletion are also presented, giving you a final chance to be sure you don't want them. ■

Looking at Sprites

by Len Lindsay 

Eventually, most COMAL 0.14 users try out the built in sprite commands. The commands are easy - once you understand how they work. But, just what is a sprite? A sprite is like a ghost. It can take different shapes and colors, and glide around the screen without affecting the graphics on the screen. It can double in size, and even *collide* with another sprite or drawing on the screen. A complete summary of all the sprite and graphics commands are printed on the back cover of this issue.

There are 8 sprites available. They can take any shape at any time. The shape is <u>not</u> a sprite, it is just the image that a sprite uses. Several sprites can use the same image at the same time. The **DEFINE** command defines a shape and automatically stores it in a special section of memory for you. Each shape you **DEFINE** is given a shape number (0-31). A string of 64 characters is used to define the shape (the last character determines if the sprite is multi-color or hi-res). You can store dozens of shapes in memory, ready for use by any of the 8 sprites. *Today Disk #17* contains a sprite editor which can produce **DATA** statements or sprite image files. Later, any of the 8 sprites can take on the shape of any of the shapes previously defined. This is done with the **IDENTIFY** command, which also turns on the sprite (so it's shape can be seen). The **HIDESPRITE** command turns off a sprite.

The sprite can be positioned anywhere on the graphics screen using X / Y co-ordinates. The screen X co-ordinates start at 0 on the left through 319 on the right. The Y co-ordinates start at 0 at the bottom through 199 at the top. When positioning a sprite on the screen, use the upper left corner of the sprite shape as the reference point.

To specify what color a sprite should be, use the **SPRITECOLOR** command. The color numbers are 0 through 15, the same as in Commodore BASIC. If you use multicolor sprites, the **SPRITEBACK** command sets the extra colors.

You also may wish to change the sprites relative size. You can double its width, height, or both with the **SPRITESIZE** command. Use **TRUE** (expanded) or **FALSE** (normal) for the **width** and the **height**.

That is enough to begin playing with sprites. Other commands you may wish to use involve detecting sprite collisions and setting the priority of a sprite in releation to the picture (data) on the screen. When checking a sprite for a collision, always use **TRUE** for the **reset flag** parameter, unless you have to test more than one sprite for a collision. In that case, use **FALSE** for the first tests, and **TRUE** only on the final test. If the data has priority, the sprite will seem to be behind it. Otherwise, the sprite looks like it is gliding over the picture drawn on the graphics screen.

If all this seems confusing, perhaps just trying the example programs will clarify sprites for you. Each program below is a full program, ready to type in and run. They are taken with permission from *Commodore 64 Graphics With COMAL*, published by Reston but now out of print (*I am quoting from my own book!*).

Note: the following programs expect the background color to be black. If this is not the case, type the following line before running the programs:

background 0

Further reference:

Easy Sprites, COMAL Today #16, page 23
Sprite Maker, COMAL Today #14, page 58
Quick Sprites, COMAL Today #13, page 18
Easy Sprites for Beginners, COMAL Today #8, page 12

more»

Target Game

This game shows you how to use the function keys during an INPUT statement to switch back and forth between the graphics and text screen. In the game, aim a gun in the bottom corner of the screen so that a shot from it will hit the target (a sprite) placed randomly on the screen.

```
dim shape$ of 64
init
repeat
 shoot
until hit
plottext 1,1,"its a hit"
//
proc init
 setgraphic 0 // turn on graphic screen
 turtlesize 3
 pencolor 12
 target
 print chr$(147), // clear screen
 print "use function key 5 to see graphics"
 print "use function key 1 to return to this"
 print
endproc init
//
proc target
 sprite'box(1) // get image
 identify 1,1 // sprite 1=image 1
 spritecolor 1,1 // white
 x:=rnd(100,199) // random x coordinate
 y:=rnd(100,199) // random y coordinate
 spritepos 1,x,y // place sprite 1 at x,y
endproc target
//
proc shoot
 moveto 1,1
 aim
 count:=0; hit:=false
 dummy:=datacollision(1,true) //clear flag
 repeat
  forward 5
  count:+5
  if datacollision(1,true) then hit:=true
 until hit or count>350
endproc shoot
//
proc sprite'box(num) closed
 dim shape$ of 64
 for x:=1 to 63 do shape$:=shape$+chr$(255)
 shape$:=shape$+chr$(0) // hi-res sprite
 define num,shape$ // define shape
endproc sprite'box
//
proc aim
 settext // back to text screen
 input "what angle ": angle
 setheading angle
 setgraphic // back to old graphic screen
endproc aim
```

Box Collision

Two sprites use the same image (a simple box). One stays on the screen while the other moves randomly. It's all over when they collide.

```
dim shape$ of 64
init
identify 2,1 // sprite 2 gets shape 1
spritepos 2,100,100 // place sprite 2 at 100,100
spritesize 2,true,true // double size
repeat
 hide'target
until spritecollision(1,true)
//
proc sprite'box(ref shape$) closed
 shape$:="" // init
 for x:=1 to 63 do shape$:=shape$+chr$(255)
 shape$:=shape$+chr$(0) // hi-res sprite
endproc sprite'box
//
proc init
 setgraphic 0 // turn on graphic screen
 hideturtle
 pencolor 12
 sprite'box(shape$)
 define 1,shape$ // define shape
 spritecolor 1,7
 spritesize 1,true,true // double size
 identify 1,1 // sprite 1, shape 1
```

more»

```
 spritepos 1,0,0
 dummy:=spritecollision(1,true)
endproc init
//
proc hide'target
 hide'x:=rnd(0,319)
 hide'y:=rnd(0,199)
 spritepos 1,hide'x,hide'y
 for x:=1 to 999 do null // pause
endproc hide'target
```

Invisible Moving Sprite

The next program shows how a sprite can be moved even while it is invisible. Hold down the space bar to see the sprite.

```
dim show$ of 1
init
x:=50; y:=10 // init coordinates
repeat
 x:=(x+5) mod 320
 for delay:=1 to 99 do null // pause
 spritepos 1,x,y
 show$:=key$
 if show$<>chr$(0) then
  identify 1,1 // sprite on
 else
  hidesprite 1 // sprite off
 endif
until show$="q" or show$="Q"
//
proc init
 pencolor 12
 setgraphic 0
 hideturtle
 sprite'box(1)
 spritecolor 1,1
 priority 1,true // sprite travels under drawing
 spritesize 1,true,false // expand width
 plottext 1,1,"hit space to see sprite, q to quit"
endproc init
//
proc sprite'box(num) closed
 dim shape$ of 64
 for x:=1 to 63 do shape$:=shape$+chr$(255)
 shape$:=shape$+chr$(0)
 define num,shape$
endproc sprite'box
```

Hopping Sprite

This program provides a hopping sprite. Watch the sprite move across the screen while changing images to give a hopping illusion.

```
dim shape$ of 64
init
repeat
 y:=100
 for walk:=1 to 319 step 5 do
  spritepos 1,walk,y+(walk mod 4)
  identify 1,walk mod 4 // change shapes
  for pause:=1 to 99 do null // pause
 endfor walk
until true=false // forever
//
proc setup
 shape$(1:64):="" // shape$ contains 64 spaces
 shape$(64):=chr$(1) // multi-color
 for x:=21 to 63 do shape$(x):=chr$(0)
 for x:=1 to 21 do shape$(x):=chr$(170)
 define 0,shape$ // define shape 0
 for x:=22 to 42 do shape$(x):=chr$(20)
 define 1,shape$ // define shape 1
 define 3,shape$ // define shape 3
 for x:=43 to 63 do shape$(x):=chr$(60)
 define 2,shape$ // define shape 2
endproc setup
//
proc init
 setgraphic 1 // set multi-color grahpic screen
 spriteback 6,2 // extra sprite colors
 spritecolor 1,1
 spritesize 1,false,false // normal size
 setup
endproc init
```

Addition Practice

This addition practice program uses a sprite to cover up the answer to the problem. Once you

more»

type the correct answer, the **PRIORITY** of the sprite is changed so the answer then is on top.

```
init
repeat
 create'problem
 show'problem
 solve'problem
until done(18,4)
pencolor 12
settext
//
proc init
 setgraphic 0 // turn on graphic screen
 hideturtle
 sprite'box(1)
 spritepos 1,8*18,8*8
 spritecolor 1,11 //dark gray
 identify 1,1 // sprite 1, shape 1
endproc init
//
proc create'problem
 num1:=rnd(0,9)
 num2:=rnd(0,9-num1)
 answer:=num1+num2
endproc create'problem
//
proc show'problem
 clear // erase graphic screen
 pencolor 1
 priority 1,false // screen data not at priority
 plottext 20*8,11*8,chr$(num1+48)
 plottext 19*8,9*8,"+"+chr$(num2+48)
 plottext 19*8,8*8,"=="
 plottext 20*8,7*8,chr$(answer+48)
endproc show'problem
//
proc solve'problem
 repeat
 until key$=chr$(answer+48)
 priority 1,true // data above sprite
endproc solve'problem
//
proc sprite'box(num) closed
 dim shape$ of 64
 for x:=1 to 63 do shape$:=shape$+chr$(255)
 shape$:=shape$+chr$(0) // hi-res sprite
 define num,shape$ // define shape
endproc sprite'box
//
func done(row,col)
 repeat
  plottext row*8,col*8,"     " // 5 spaces
  case key$ of
  when "y","Y"
   return false
  when "n","N"
   return true
  otherwise
   plottext row*8,col*8,"more?"
  endcase
 until true=false // forever
endfunc done
```

Guess My Number (colors)

This guess my number game uses color to tell you how close you are to the correct number. The number is behind the box (a sprite) all the time. After you guess it, the **PRIORITY** is changed to allow the number to be seen.

```
init
game
//
proc init
 pencolor 12
 setgraphic 0 // turn on hi-res graphic screen
 hideturtle
 sprite'box(1)
 identify 1,1
 spritecolor 1,0
 spritesize 1,false,false // nowmal size
 priority 1,false // sprite on top of data
endproc init
//
proc game
 number:=rnd(0,9)
 spritepos 1,9*8,12*8
 plottext 10*8,10*8,chr$(number+48)
 plottext 1,1,"guess my number (0-9):"
 repeat
```

more»

```
   guess:=get'digit
   if guess=number then
    priority 1,true // data on top of sprite
    plottext 1,1,"you guessed my number "
   elif abs(guess-number)=1 then
    spritecolor 1,red
   elif abs(guess-number)=2 then
    spritecolor 1,orange
   elif abs(guess-number)<=4 then
    spritecolor 1,yellow
   elif abs(guess-number)<=8 then
    spritecolor 1,white
   else
    spritecolor 1,black
   endif
  until guess=number
endproc game
//
proc sprite'box(num) closed
 dim shape$ of 64
 for x:=1 to 63 do shape$:=shape$+chr$(255)
 shape$:=shape$+chr$(0) // hi-res sprite
 define num,shape$ // define shape
endproc sprite'box
//
func black
 return 0
endfunc black
//
func white
 return 1
endfunc white
//
func red
 return 2
endfunc red
//
func yellow
 return 7
endfunc yellow
//
func orange
 return 8
endfunc orange
//
func get'digit closed
 dim c$ of 1
 repeat
  c$:=key$
 until c$ in "0123456789"
 digit:=ord(c$)-48
 return digit
endfunc get'digit
```

Target Sighting Demo

This program uses a sprite for target sighting. The cursor keys move it around the screen.

```
init
process
//
proc init
 pencolor 12
 input "moving speed (1-9): ": speed
 if speed<1 then speed:=1
 if speed>9 then speed:=9
 setgraphic 0 // turn on hi-res graphic screen
 hideturtle
 define'shap(1)
 identify 1,1 // sprite 1, shape 1
 xpos:=100;  ypos:=100
 spritecolor 1,1
endproc init
//
proc process
 repeat
  spritepos 1,xpos,ypos
  case key$ of
  when chr$(145) // cursor up
   ypos:+speed
  when chr$(17) // cursor down
   ypos:-speed
  when chr$(157) // cursor left
   xpos:-speed
  when chr$(29) // cursor right
   xpos:+speed
  when chr$(133) // f1
   spritesize 1,false,false // normal size
  when chr$(134) // f3
   spritesize 1,true,true // double size
  when "q","Q",chr$(136) // f7
```

more»

```
    end
   otherwise
    null
   endcase
  until true=false // forever
endproc process
//
proc define'shap(n) closed
 restore  // watch out for other data
 dim shape$ of 64
 for temp:=1 to 64 do
  read temp'data
  shape$(temp):=chr$(temp'data)
 endfor temp
 define n,shape$
 data 255,0,255,255,0,255,192,0,3,192,0,3,192,0,3
 data 192,0,3,192,0,3,0,24,0,0,24,0,0,24,0,0
 data 255,0,0,24,0,0,24,0,0,24,0,192,0,3,192,0
 data 3,192,0,3,192,0,3,192,0,3,255,0,255,255,0,255
 data 0
endproc define'shap
```

Sprite Cake

Use the function keys to change the colors on this simple multicolor sprite.

```
init
plottext 1,1," f1=top  f3=middle"
plottext 1,160," f5=bottom  f7=quit"
process
//
proc init
 setgraphic 0
 hideturtle
 sprite'cake(1)
 color:=1; color1:=2; color2:=7
 spriteback color1,color2 // extra colors
 spritecolor 1,color // sprite 1, shape 1
 identify 1,1
 spritepos 1,100,160
 spritesize 1,true,true
endproc init
//
proc process
 repeat
  done:=false
  case key$ of
  when chr$(133) // f1
   color1:=(color1+1) mod 16
  when chr$(134) // f3
   color2:=(color2+1) mod 16
  when chr$(135) // f5
   color:=(color+1) mod 16
  when chr$(136) // f7
   done:=true
  otherwise
   null
  endcase
  spriteback color1,color2 // change colors
  spritecolor 1,color // change colors
 until done
endproc process
//
proc sprite'cake(num) closed
 dim shape$ of 64
 shape$(1:64):="" // shape$ is 64 spaces
 shape$(64):=chr$(1) // multi-color sprite
 for layer:=1 to 21 do
  shape$(layer):=chr$(85)
  shape$(layer+21):=chr$(255)
  shape$(layer+42):=chr$(170)
 endfor layer
 define num,shape$ // define shape
endproc sprite'cake
```

Target Practice

Instead of moving targets, a ball rolls accross the screen. Hit the space bar to shoot.

```
init
for trys:=1 to 3 do game // 3 games
pencolor 12
settext
//
proc init
 setgraphic 0 // turn on hi-res graphics
 plottext 1,1,"hit space bar to shoot"
 sprite'box(1)
 for x:=0 to 6 do
  identify x,1 // sprite x, shape 1
```

more»

```
  spritesize x,true,true // double size
  spritecolor x,x // sprite x, color x
  spritepos x,x*6*8,194
 endfor x
 spritecolor 6,0
 sprite'dot(2)
 identify 7,2 // sprite 7, image 2
 spritesize 7,true,true // double size
 spritecolor 7,1
 spritepos 7,1,1
 moveto 0,199
 pencolor 7
 drawto 319,199
 moveto 0,195
 drawto 319,195
 fill 1,196
endproc init
//
proc game
 y:=10 // pixel row for ball to roll down
 for x:=1 to 319 step 2 do
  spritepos 7,x,y
  if key$=" " then // space bar?
   dummy:=spritecollision(7,true); movey:=0
   repeat
    movey:+3
    spritepos 7,x,y+movey
   until spritecollision(7,true)
   for s:=1 to 5 do
    if spritecollision(s,false) then spritecolor s,0
   endfor s
   while key$>chr$(0) do null //clear keyboard
  endif
 endfor x
endproc game
//
proc sprite'dot(num) closed
 dim shape$ of 64
 for x:=1 to 64 do shape$(x):=chr$(0)
 shape$(1):=chr$(192); shape$(4):=chr$(192)
 define num,shape$ // define shape
endproc sprite'dot
//
proc sprite'box(num) closed
 dim shape$ of 64
 for x:=1 to 63 do shape$:=shape$+chr$(255)
 shape$:=shape$+chr$(0) // hi-res sprite
 define num,shape$ // define shape
endproc sprite'box
```

Word Hider

The next program shows an interesting use of sprites. The sprites begin with their size expanded and thus cover the words. However, hitting the appropriate keys shrink the sprite size, revealing the words.

```
max:=4 // number of boxes
dim word$(1:max) of 6, used#(1:max)
dim reply$ of 1
init
repeat
 setup
 play
until done(1,1)
pencolor 12
settext
//
proc init
 setgraphic 0
 pencolor 12
 hideturtle
 sprite'rec(1)
 for number:=1 to max do
  identify number,1 // sprite number, shape 1
  spritesize number,true,true // double size
  spritecolor number,number
  priority number,false // sprite on top of data
 endfor number
endproc init
//
proc sprite'rec(num) closed
 dim shape$ of 64
 shape$(1:64):="" // shape$ is 64 spaces
 shape$(64):=chr$(0) // hi-res sprite
 for layer:=1 to 21 do
  shape$(layer):=chr$(255)
  shape$(layer+21):=chr$(255)
  shape$(layer+42):=chr$(0)
 endfor layer
 define num,shape$ // define shape
```

more»

```
endproc sprite'rec
//
proc get'word(ref w$)
 restore
 for trys:=1 to rnd(1,26) do read w$
endproc get'word
//
proc setup
 clear // erase graphic screen
 pencolor 12
 for x:=1 to max do
  spritesize x,true,true // double size
  get'word(word$(x))
  spritepos x,x*(7*8),8*20
  plottext x*8*7,17*8,word$(x)
  plottext x*8*7+8*3,21*8,word$(x)(1)
 endfor x
endproc setup
//
proc play
 hit:=0
 for temp:=1 to max do used#(temp):=false
 repeat
  plottext 1,1,"hit each of the letters please"
  reply$:=key$
  for num:=1 to max do
   if word$(num)(1)=reply$ and not used#(num)
    spritesize num,true,false // shrink height
    used#(num):=true; hit:+1
   endif
  endfor num
 until hit=max
 plottext 1,1,"                                "
 // plot 32 spaces
endproc play
//
func done(row,col)
 repeat
  plottext row*8,col*8,"     " // 5 spaces
  case key$ of
  when "y","Y"
   return false
  when "n","N"
   return true
  otherwise
   plottext row*8,col*8,"more?"
  endcase
 until true=false // forever
endfunc done
//
data "ant","bear","camel","duck","elk"
data "fox","goat","hyena","iguana","jaguar"
data "koala","lion","monkey","newt","otter"
data "panda","quail","raven","skunk","tiger"
data "urchin","vixen","walrus","x","yak","zebra"
```

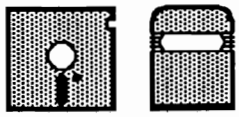
Shape Definition Proc Maker

This program creates a customized procedure including data statements of an image from an image data file (file names starting with *imag.*) Sample image files are on *Today Disk #17*. Just RUN this program and enter the name of the file after the imag. Your new shape defining procedure is listed on the screen. You then cursor to the top and hit «*return*» 17 times to enter the procedure into program memory. Somewhere in your main program include a line to execute this procedure like:

```
define'shap(1)
```

```
dim text$ of 64, name$ of 18
input chr$(147)+"image file name: imag.":name$
open file 2,"imag."+name$,read
read file 2: text$
close
print chr$(147),"9926 proc define'shap(n) closed"
print "9927  restore//watch for other data"
print "9928  dim shape$ of 64"
print "9929  for temp=1 to 64 do"
print "9930   read temp'data"
print "9931   shape$(temp)=chr$(temp'data)"
print "9932  endfor temp"
print "9933  define n,shape$"
for x:=0 to 63 step 8 do
 print 9934+x;"data";ord(text$(x+1)),
 for y:=2 to 8 do print ",",ord(text$(x+y)),
 print
endfor x
print "9999 endproc"
print "!add to program: define'shap(1)" ■
```

Calculate PI



by Jack Baldrige

There's not really much reason to calculate the value of **PI** to any great accuracy. I've read somewhere that with a value to 25 digits, you could express the diameter of the earth to the nearest atom.

Still, before the advent of computers, somebody calculated the value of **PI** to over 500 places by mechanical means. The first time it was done with a computer was with Eniac in 1949. It calculated **PI** to 2037 places in 70 hours. Even the C64 does much better than that today. It took the C64 less than 14 hours to get 2500 places using COMAL.

Today, there is one practical use for these calculations. If you get a few hundred thousand digits on a new computer, you know that it's reliable (in integer operations, anyway). The latest such job I have figures on was a calculation in France on a CDC 6600. It got 500,000 digits (and checked them with a different algorithm) in 44 hours, 45 minutes. I know, however, that the same trick has been done with supercomputers such as the Cray, so heaven knows the latest number of digits.

The most common algorithm, the one I used, was developed by John Machin about 1700:

pi = 16*arctan(1/5)-4*arctan(1/239)

The arc tangent, from Gregory's series, is:

arctan(1/x) = 1/x-1/(3*x^3)+1/(5*x^5)-....

As you can imagine, the arctan series converges quickly when x=5 and goes like fury for x=239!

Admittedly, there isn't a very good reason to do a calculation like this, but at least, you get a result that's a little more accurate than you get with a pocket calculator. *Calculate'pi* is on both sides of *Today Disk #17*.

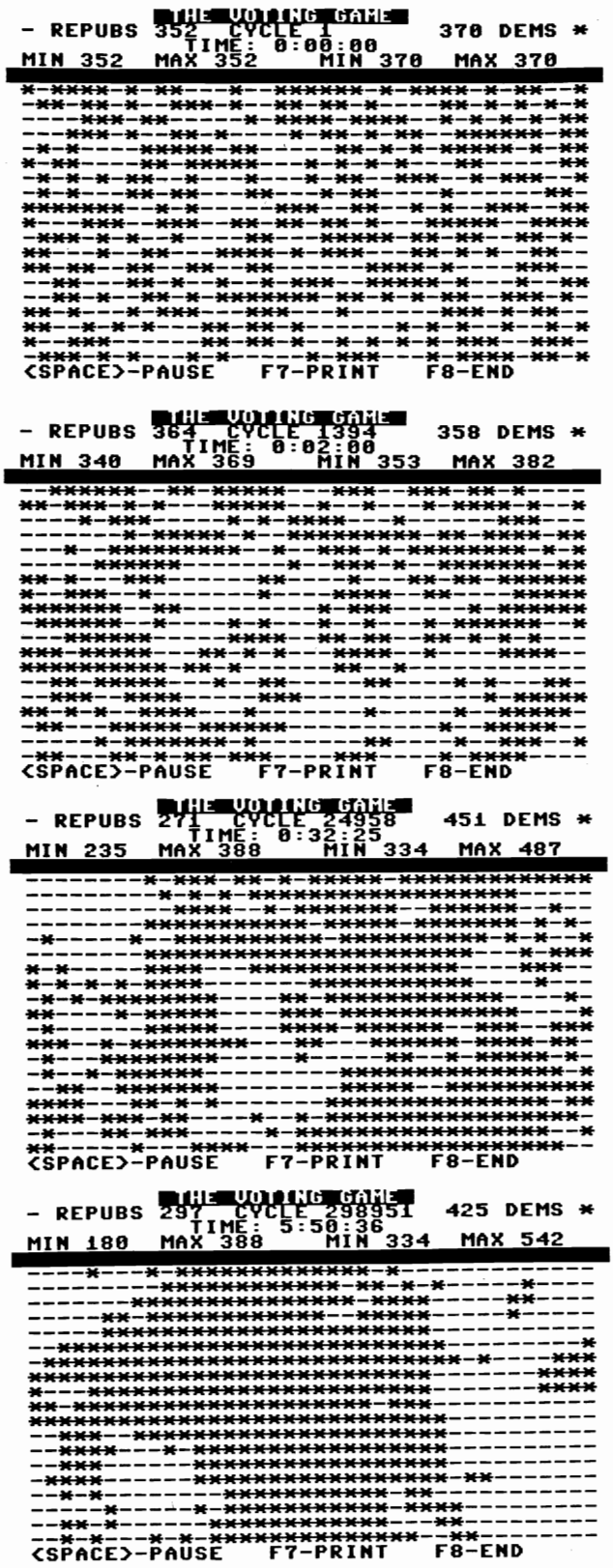
```
3.14159265358979323846264338327950288419716939937510582097494459
23078164062862089986280348253421170679821480865132823066470938446095505822317253594081284811174
```

```
dim q$ of 1 // 2.0 users must add a () after
print chr$(147) // array names as parameters
print "   high precision pi calculations"
print
input " minimum number of digits? ": digit#
max#:=digit# div 4+2
digit#:=max#*4-4; array'size#:=max#
hcopy#:=false
input " output to printer? n"+chr$(157): q$
if q$="Y" or q$="y" then hcopy#:=true
setzero
header
dim pi'array#(max#), partial#(max#)
dim n$ of digit#, b$ of 10
b$:="0123456789"
series(pi'array#,5,max#) // 2.0 add ()
multiply(pi'array#,4,max#) // 2.0 add ()
series(partial#,239,max#) // 2.0 add ()
subtract(max#,partial#,pi'array#) // 2.0 add ()
multiply(pi'array#,4,max#) // 2.0 add ()
print
if hcopy# then
 select output "lp:"
 header
endif
print'pi(max#,pi'array#,n$,b$) // 2.0 add ()
j:=int(seconds*100+.5)/100
print "time is";j;"seconds"
select output "ds:"
end
//
proc series(ref array#(),f,max#) closed
 dim power#(max#), temp#(max#)
 power#(max#):=1000; array'size#:=max#
 divide(power#,f,array'size#,max#) //2.0 add ()
 sign:=1; divisor:=3; count:=0
 for k:=1 to max# do array#(k):=power#(k)
 while array'size#>0 do
```

more»

```
  count:+1
  divide(power#,f*f,array'size#,max#)//2.0 add ()
  sign:=-sign
  for j#:=1 to max# do temp#(j#):=power#(j#)
  divide(temp#,divisor,array'size#,max#)//2.0add()
  divisor:+2
  if sign>0 then
   add(array'size#,temp#,array#) // 2.0 add ()
  else
   subtract(array'size#,temp#,array#)//2.0 add ()
  endif
 endwhile
endproc series
//
proc divide(ref array#(),divisor,ref array'size#
 ,max#) closed // wrap line
 remainder:=0; array'size#:=max#
 while array#(array'size#)=0 do
  array'size#:-1
  if array'size#=0 then return
 endwhile
 for i:=array'size# to 1 step -1 do
  term:=10000*remainder+array#(i)
  array#(i):=term div divisor
  remainder:=term mod divisor
 endfor i
 if array#(array'size#)=0 then array'size#:-1
endproc divide
//
proc add(ref array'size#,ref tarray#(),ref
 sarray#()) closed // wrap line
 carry:=0
 for k#:=1 to array'size# do
  sum:=tarray#(k#)+sarray#(k#)+carry
  sarray#(k#):=sum mod 10000
  carry:=sum div 10000
 endfor k#
 if carry then sarray#(array'size#+1):+carry
endproc add
//
proc subtract(ref array'size#,ref termarray#(),ref
 diffarray#()) closed // wrap line
 borrow:=0
 for k#:=1 to array'size# do
  diff:=diffarray#(k#)-termarray#(k#)+borrow
  borrow:=diff div 10000
  diffarray#(k#):=diff mod 10000
 endfor k#
 if borrow then diffarray#(array'size#+1):-1
endproc subtract
//
proc multiply(ref array#(),factor#,max#) closed
 carry:=0
 for j#:=1 to max# do
  product:=factor#*array#(j#)+carry
  array#(j#):=product mod 10000
  carry:=product div 10000
 endfor j#
endproc multiply
//
proc print'pi(max#,ref array#(),ref n$,ref b$
 ) closed // wrap line
 dim p$ of 4
 print
 for k#:=max# to 2 step -1 do
  string(array#(k#),p$,b$)
  n$:=n$+p$
 endfor k#
 print "pi equals";n$
 for k#:=1 to 3 do print
endproc print'pi
//
proc header
 print " calculating pi to";digit#;"digits..."
endproc header
//
proc string(ref n#,ref p$,ref b$) closed
 hi:=n# div 100; lo:=n# mod 100
 p$:=b$(hi div 10+1); p$:=p$+b$(hi mod 10+1)
 p$:=p$+b$(lo div 10+1); p$:=p$+b$(lo mod 10+1)
endproc string
//
func seconds closed
 j:=256*256*peek(160)+256*peek(161)+peek(162)
 return j/60
endfunc seconds
//
proc setzero closed
 poke 160,0
 poke 161,0
 poke 162,0
endproc setzero ■
```

The Voting Game



by Bob McCauley

The *voting'game* is a COMAL 2.0 simulation of a problem discussed in the April 1985 *Scientific American* Computer Recreations column written by A. K. Dewdney. Assume you have a random population of Republicans and Democrats (Liberals and Tories, Democrats and Socialists, whatever). Regularly one voter is selected at random, one of his neighbors is selected at random, and he converts to his neighbor's party.

This simulation creates a rectangular array, populates it randomly with republicans and democrats, and then proceeds. There are eight neighbors (the edges wrap in both directions). As time progresses you first see the parties start to group. Then you see the groups shift and move around the screen, with first one and then the other gaining the majority.

The ultimate question - will democracy survive or will one party be overwhelmed and die out?

The program has arrays of 20x10, 30x15, or 38x19. The smallest sometimes gives a solution, and the big one has run several million cycles without one party dying out. There are provisions to pause, print the display, and end. Pause provides a readout of voter populations and cycle count. Those also can be continuously displayed, but that slows the program down.

I hope to run this on a Zenith Z-248 sometime, and the increase to 8 MHz should provide a fascinating display. The movement of the patterns should be easily discernable.

The companion program, *antivoter*, does the opposite of the *voting'game*. It changes the voter's party to the *opposite* of the neighbor's party. You'll see the different result. Both programs are on *Today Disk #17*. We may be able to convert the game to COMAL 0.14 for *Today Disk #17*. ■

Learning Subtraction Part II

by Len Lindsay 

Learning subtraction takes practice. The more you practice the better you get. Last issue I provided a program that would flash subtraction problems on the screen. Now let's produce some practice sheets. As before, this program is flexible, allowing you to set the highest numbers to use. If you need an answer sheet, that option is available. You also can get multiple copies of the same practice sheet so that several people can participate - great for time tests. In case you don't have a printer, I even allow the sheets to print on the screen.

If you looked ahead at the program listing, you noticed that it is very readable again. The program works with COMAL 0.14 or 2.0 without any changes.

With this program I offer some advice: don't reinvent the wheel. Make it easy on yourself. COMAL is on your side. In COMAL, you give a name to all the routines that you write. In this issue's program, I use three routines from last issue: **erase'screen**, **get'num**, and **yes'no**. I can steal them directly from the other program without retyping them! Here is a brief explanation:

COMAL has two ways to store program lines on disk. **SAVE** will store the complete program in tokenized (compressed) form:

SAVE "NAME"

That is how you usually will do it. COMAL also can store program lines on disk in ASCII format as a sequential file:

LIST "NAME.LST"

COMAL uses these ASCII files to provide you with a merge capability. You can store common routines on disk in ASCII format, and then merge them into future programs. This format also is compatible with many word processors, allowing you to transfer a program into a text file for an article that you may be writing. No need to retype the program in the wordprocessor.

Now, I must explain one fine point about merging routines from program to program. COMAL 0.14 merges the lines from the ASCII disk file using the same exact line numbers that were used when the routine was listed to disk. If your current program includes similar line numbers, the new lines being merged will overwrite the current ones. You probably do not want this to happen. (If you have the COMAL 2.0 cartridge, you don't have to worry about this. The **MERGE** command renumbers the lines being merged automatically).

The way I avoid line number problems is to make sure the line numbers in my ASCII **LIST** files are over 9000. Future programs I write will not have line numbers that high. It is easy to do. Since we will use three routines from last issue, lets store them on disk in ASCII format, ready to merge with future programs. First **LOAD** the program from last issue:

LOAD "SUBTRACT"

Now, let's renumber it so that all its lines are over 9000:

RENUM 9001,1

Now the program lines follow the sequence: 9001, 9002, 9003, etc. Next list the program and note the starting and ending numbers for the routines we want to store for future use:

LIST

Hold down the «*ctrl*» key to slow down the listing. Hit the «*space*» bar to pause the listing. Hit the «*space*» bar again to restart the listing.

more»

You can store a range of lines to disk in ASCII format, rather than the entire program. That is what we will do now:

LIST *9016-9025* **"FUNC.YES'NO"**
LIST *9037-9043* **"FUNC.GET'NUM"**
LIST *9101-9104* **"PROC.ERASESCREEN"**

[These line numbers match the subtract program entered exactly as it appeared in COMAL Today #16 or on Today Disk #16, 0.14 side. Make sure they are correct for your version of the program.]

The routines now are ready to be merged into future programs. We will do that shortly.

Now, back to our current program. We want to print subtraction problem sheets. Let's get started with the main program. First type NEW to get rid of any previous programs, then **AUTO** to allow COMAL to provide the line number for you:

```
NEW
AUTO
```

Now type in these lines:

```
erase'screen
init
repeat
  problem'sheets
until yes'no("Do you want another")=false
```

OK, now stop the **AUTO** mode by hitting the «*return*» key twice (cartridge users, hit the «*stop*» key instead of the second «*return*»). Now, let's merge those three routines into this program. Make sure the disk in the drive has the routines on it. Then enter these commands:

```
ENTER "PROC.ERASESCREEN"
RENUM
ENTER "FUNC.GET'NUM"
RENUM
ENTER "FUNC.YES'NO"
RENUM
```

(Cartridge users should use the **MERGE** command instead of **ENTER**).

Now, **LIST** the program you have so far to see the routines we just merged:

```
erase'screen
init
repeat
  problem'sheets
until yes'no("Do you want another")=false
//
proc erase'screen
  print chr$(147),
endproc erase'screen
//
func get'num(prompt$,default)
  print prompt$;default
  print chr$(145), //cursor up
  input prompt$+"?": number
  return number
endfunc get'num
//
func yes'no(question$) closed
  dim reply$ of 1
  input question$+"? ": reply$
  if reply$ in "yY" then
    return true
  else
    return false
  endif
endfunc yes'no
```

Use the **AUTO** system to provide line numbers as you type in the rest of the program (continue with proc INIT on page 34). Be careful: the plain AUTO command will start with line 0010 again. You want to start with line 0270 so use the AUTO command with a starting line number specified:

```
AUTO 270
```

more»

You probably noticed a few more features of COMAL as you typed in the program. The first was the arrays dimensioned in the **init** routine. BASIC forces you to start your array index with 0. COMAL lets you use any number you like. I chose to start with 1, since I prefer to start counting with 1 rather than 0. If you don't specify a starting index number, COMAL will use 1 by default. Thus both my arrays use 1 to start the index, even though I only specified it for the first one.

I hope you also noticed how easy it is to direct output from a program to your printer. You simply **SELECT** the printer as the output location (**"lp:"** stands for Line Printer). To return the output to the screen use **"ds:"** (which stands for Data Screen). This also makes it easy to list your program to your printer:

```
SELECT "LP:"
LIST
```

Did you also notice that the same routine (named **printout**) prints the problem sheets with or without answers? Look at the first line of the procedure:

```
PROC PRINTOUT(WITH'ANSWERS)
```

With'answers is a parameter. COMAL lets you pass information in and out of your procedures and functions using parameters. Remember, we did the same thing with the **get'num** and **yes'no** functions. This time we just use the value of **TRUE** or **FALSE** to let our **printout** routine know whether or not to print the answers. (Yes, COMAL knows what the words **TRUE** and **FALSE** mean).

This program also takes advantage of COMAL's ability to disable the «*stop*» key:

```
TRAP ESC -
```

This tells COMAL not to stop the program if someone hits the «*stop*» key. Instead, COMAL will set the system variable named **ESC** to be **TRUE**. We check the value of **ESC** in the procedure **next'line**. If it is **TRUE**, then someone hit the «*stop*» key, so we call our procedure named **halt**. This returns the output back to the screen and ends the program. This is important. If you stop the program while it is printing on the printer, the output location remains on the printer. If you then gave a command like **LIST**, it would list to your printer rather than the screen.

There is a prompt near the end of the **printout** procedure telling the user to *remove sheet, hit return when ready*. This allows users set up the paper in the printer for each sheet. If you own a printer which handles automatic form feeds, you may wish to have this work done for you. Change the line to the following:

```
PRINT CHR$(12) // form feed
```

If you don't wish to type in the program, it is available on *Today Disk #17*.

```
erase'screen
init
repeat
 problem'sheets
until yes'no("Do you want another")=false
//
proc erase'screen
 print chr$(147),
endproc erase'screen
//
func get'num(prompt$,default)
 print prompt$;default
 print chr$(145), // cursor up
 input prompt$+"?": number
 return number
endfunc get'num
//
func yes'no(question$) closed
 dim reply$ of 1
```

more»

```
 input question$+"? ": reply$
 if reply$ in "yY" then
  return true
 else
  return false
 endif
endfunc yes'no
//
proc init
 max'top:=get'num("Max top number",19)
 max'bot:=get'num("Max bottom number",9)
 max'answer:=get'num("Max answer number",10)
 num'accross:=get'num("Problems per row",10)
 num'down:=get'num("Number of rows",10)
 dim problems'top(num'accross,num'down)
 dim problems'bot(num'accross,num'down)
 dim reply$ of 1
 num'copies:=get'num("Number copies",1)
 answer'sheet:=yes'no("Do you want an answer
 sheet") // wrap line
 printer:=yes'no("Do you want it on the
printer")
endproc init
//
proc problem'sheets
 print "Thinking....."
 for rows:=1 to num'down do
  for cols:=1 to num'accross do
   get'problem
  endfor cols
 endfor rows
 trap esc-
 if printer then select output "lp:"
 for sheets:=1 to num'copies do
  printout(false) //no answers
 endfor sheets
 if answer'sheet=true then printout(true)
 select output "ds:" //back to screen
 trap esc+
endproc problem'sheets
//
proc get'problem
 problems'top(cols,rows):=rnd(1,max'top)
 low:=problems'top(cols,rows)-max'answer
 if low<0 then low:=0
 high:=problems'top(cols,rows)
 if high>max'bot then high:=max'bot
 problems'bot(cols,rows):=rnd(low,high)
endproc get'problem
//
proc printout(with'answers)
 for rows:=1 to num'down do
  for cols:=1 to num'accross do
   show(problems'top(cols,rows))
  endfor cols
  next'line
  for cols:=1 to num'accross do
   show(-problems'bot(cols,rows))
  endfor cols
  next'line
  for cols:=1 to num'accross do showbar
  next'line
  if with'answers=true then
   for cols:=1 to num'accross do
    answer:=problems'top(cols,rows)-prob
    lems'bot(cols,rows) // wrap line
    show(answer)
   endfor cols
  endif
  next'line
  if rows<num'down then next'line
 endfor rows
 if printer=true then
  input "remove sheet, hit return when
  ready":reply$ // wrap line
 else
  ask'next
 endif
endproc printout
//
proc show(number)
 print using "-##": number,
 spacing
endproc show
//
proc showbar
 print " ==",
 spacing
endproc showbar
//
proc ask'next
 input " Hit return when ready...": reply$
```

more»

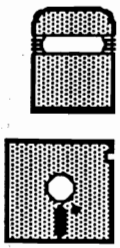
```
 next'line
endproc ask'next
//
proc halt
 while esc do null
 trap esc+
 select output "ds:" // back to screen
 print "thank you."
 end
endproc halt
//
proc next'line
 print  // carriage return
 if esc=true then halt // stop key hit
endproc next'line
//
proc spacing  // between problems
 if printer then print "  ", // 2 spaces
endproc spacing
```

Sample Answer Sheet

```
19 15  7  1 17 13  6  6 10 15
-9 -6 -2 -1 -9 -3 -3 -1 -3 -9
== == == == == == == == == ==
10  9  5  0  8 10  3  5  7  6

 3 16 18  2 11 17  8 13  9 18
 0 -7 -9  0 -6 -7 -7 -4 -5 -8
== == == == == == == == == ==
 3  9  9  2  5 10  1  9  4 10

13 17 18  6 18 18  1 14 15 15
-8 -7 -9 -6 -9 -9  0 -5 -5 -6
== == == == == == == == == ==
 5 10  9  0  9  9  1  9 10  9

 4 15 12  6 14 17  7  2  3  5
-1 -7 -3 -2 -6 -7 -6 -1 -2 -2
== == == == == == == == == ==
 3  8  9  4  8 10  1  1  1  3
```

■

Pressure Tester



by Dick Klingens, Holland Users Group

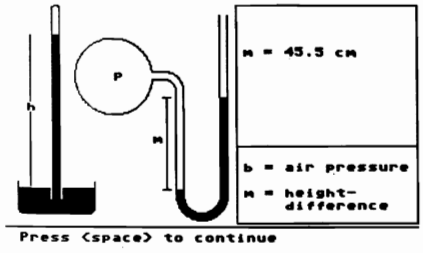
Baro&mano and *pressure* are programs which are used in Holland in a first course of physics at the Duno College at Doorwerth. They are for practice and drill.

A series of simple exercises on pressure measuring is presented on the screen and the student has to solve the problems with help of a hand held calculator. The program only checks the student's answers. No other help is provided in the program.

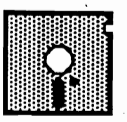
Each exercise starts with a picture of a Torricelli pipe with some data, such as the air pressure or the gas pressure in a open or closed manometer. After that, the student has to do calculations (such as convert pressure to the height of a column of mercury) and type in the answers.

The program is not as user friendly as possible, but because it was written in COMAL, one can change it in an easy way. The original program was written with version 0.14. The author of the program, Paul van Leeuwen, implemented procedures such as **print'at** and **input'at** which emulate the COMAL 2.0 commands.

He started his educational program development with the program *pressure*. Both versions, 0.14 and 2.0, are on *Today Disk #17*. The 0.14 programs have Dutch names for identifiers. The COMAL 2.0 version is in English. ■



Russian Roulette



Dear Captain COMAL,

I received my Programmers Paradise Package a little while ago, and stayed up till 3:00 AM reading the books and issues of *COMAL Today* (not recommended, as one has a tendency to act slightly stupid the next day).

I love COMAL! It's the greatest language I've ever seen. I'm familiar with several dialects of BASIC, assembly language, and BASIC extensions. COMAL is the best! Once you unlearn all those bad habits you pick up using BASIC, it's truly a programmers paradise!

I've just finished converting a simple BASIC game to COMAL, it's the game Russian Roulette. In a back issue of *COMAL Today*, one fellow bemoaned the lack of games in COMAL, and I thought, hey, there's tons of games out there that are fairly short (thus no worry about memory limitations), which are written in BASIC. Once you manage to make some sense of the author's spaghetti code, the conversion to COMAL is pretty simple. Most can be found in books of the *101 BASIC Games* type. And some really are pretty neat. With COMAL's graphics commands and ease of extendability (using PROCs and FUNCs), many could be greatly improved.

Granted, this lacks the originality of writing your own games, but I've found that (for me at least), it helps you learn COMAL (and also lets you see how much better COMAL is than any version of BASIC). - Mark Skopinsky

```
// delete "0:russian'roulette"
// by mark skopinsky
// save   "0:russian'roulette"
//
dim choice$ of 1
trap esc-
print "This is a game of Russian Roulette."
print "Here is a revolver."
print "If you spin 10 times without blowing"
print "your brains out, you win."
instructions
//
proc instructions
 print "Press 'S' to spin the chamber"
 print "and pull the trigger."
 print "Press 'Q' to quit the game.",chr$(13)
 times=0
 input'routine
endproc instructions
//
proc input'routine
 input "Your choice :": choice$
 case choice$ of
 when "s","S"
  random'routine
 when "q","Q"
  chicken
 otherwise
  game'over
 endcase
endproc input'routine
//
proc random'routine
 number:=rnd(1,100)
 if number>83 then
  bang
 else
  click
 endif
endproc random'routine
//
proc click
 print "- CLICK -"
 times:=times+1
 if times=10 then win
 input'routine
endproc click
//
proc bang
 print "      BANG!!!! You're dead!"
 print "Condolences will be sent to"
 print "your next of kin."
 print "Next victim...."
 instructions
endproc bang
```

more»

```
//
proc win
 print "YOU WIN!!!!"
 print "Let someone else blow his brains out."
 instructions
endproc win
//
proc chicken
 print chr$(13),"CHICKEN!!!!"
 game'over
endproc chicken
//
proc game'over
 end
endproc game'over
```

This letter and program are special for several reasons. First, we want to congratulate Mark for the close attention he paid to the newsletter. He didn't claim to be original, but we can't help but feel good when we hear all our favorite cliches about COMAL in one letter. Amazingly, we recieved his letter within ten days of when we first sent him COMAL. What is even more suprising is the quality of this program which seems to have been written by an absolute beginner.

This program is a classic example of what we mean when we say that COMAL is a *structured* language. Note that every task is completed by a small procedure. Each one is small enough to fit on the screen. The **input'routine** shows how this can be done. You are given the choice of continuing the game or quitting. Rather than separating the choices by the code required to play the game, the procedure calls a second procedure to perform that task. This makes the menu option very readable. When someone needs to know how the game is played, he can look into the next level of the code (the **random'routine** procedure).

Other examles of structure can be seen inside the procedures. The **click** procedure contains a one line IF structure to see if you won the game. The menu from the **input'routine** is a perfect example of the CASE structure. The **random'routine** uses the IF - THEN - ELSE - ENDIF structure to distinguish between the two options of living or dying after you shoot the gun. These are powerful programming options to be mastered. COMAL allows them to be used so naturally that maybe we shouldn't be surprised to see them from a beginner.

As good as this program is, it is not perfect. A problem in this program is the indirect recursion. For example, **instructions** calls **input'routine** which calls **random'routine** which calls **click** which calls **win** which calls **instructions**. Each time COMAL calls a procedure, information must be stored to tell COMAL what to do when the procedure ends. Normally, the space used for this information is returned to the free memory section when the procedure ends. However, in this example, procedure **instructions** never ends. COMAL is inside all five procedures at once. Recursion allows this number to increase each time **instructions** is called again. Eventually this may lead to an out of memory error. A better way to do this would be put the procedure to play the game inside a loop. To play the game again, you merely repeat the loop. This method allows the procedures to end normally so they don't continually steal more memory.

Russian'roulette is on *Today Disk #17*. It works in both COMAL 0.14 and COMAL 2.0. We liked the program so much that we didn't have the heart to make the changes suggested above. We did, however, make a few changes in the PRINT statements to make them easier to list in the newsletter. *[This type editing is normal].* ■

Crystal Ball

by Mark Skopinsky

Ever wanted to ask a Gypsy fortune-teller to look into your future? Well, this program is the nearest thing to a real Gypsy with her crystal ball. You can *ask* the program questions on almost any topic, and have it *tell your fortune.*

This program is excellent to use when people ask what your computer can do, or with people unfamiliar with computers. Both the 2.0 and 0.14 versions work in the same manner, with the exception of extra procedures in the 0.14 version to draw circles (0.14 lacks 2.0's **CIRCLE** and **ARC** commands, and 0.14 doesn't allow you to plot text on a Multi-Color graphics screen). See Mindy Skelton's *Graphics Primer* for the circle procedure. *Crystal Ball* is on *Today Disk #17*. The 0.14 version is listed below.

```
dim prediction$ of 35, again$ of 1
print chr$(147)
background 6
border 6
title'screen
instructions
//
proc main
 subject
 random'number
 tell'answer(number)
 go'again
endproc main
//
proc title'screen
 setgraphic 0
 hideturtle
 pencolor 1
 border 6
 background 6
 make'circle(40,119,110)
 moveto 170,128
 moveto 185,105
 moveto 130,77
 drawto 115,55
 drawto 205,55
 drawto 190,77
 plottext 90,160,"comal crystal ball"
 pencolor 7
 plottext 103,25,"(press any key)"
 pencolor 1
 while key$=chr$(0) do null
endproc title'screen
//
proc instructions
 settext
 print chr$(147)
 print "welcome! you may ask the crystal ball"
 print "questions about almost anything. all"
 print "you need to do is tell it the subject"
 print "of your question. it can tell what "
 print "your exact question is just by the"
 print "subject (after all, crystal balls are"
 print "magic)!"
 main
endproc instructions
//
proc subject
 print
 print "the subjects you can ask questions"
 print "about are:"
 print
 print "1) money  2)job  3)health"
 print "4) school 5)love 6)quit"
 print
 input "which subject? ": reply
 if reply=6 then
  end'program
 elif reply>6 then
  print
  print "sorry, i didn't get that. try again."
  subject
 endif
endproc subject
//
proc random'number
 number:=rnd(1,5)
endproc random'number
//
proc tell'answer(answer)
 if answer=1 then
  prediction$:="certainly!"
```

more»

```
 elif answer=2 then
  prediction$:="the future is too cloudy, ask
  later." // wrap line
 elif answer=3 then
  prediction$:="never!"
 elif answer=4 then
  prediction$:="the crystal cannot give an
  answer." // wrap line
 elif answer=5 then
  prediction$:="maybe...."
 endif
 print
 print "hmm, let's see now..."
 timer
 print "the mists in the crystal are parting,"
 print "and the answer is:"
 timer
 print
 pencolor 7
 print prediction$
 pencolor 1
endproc tell'answer
//
proc go'again
 print
 print "do you wish to ask another question of"
 print "the magical comal crystal ball??"
 input "(y/n): ": again$
 if again$ in "nN" then
  end'program
 elif again$ in "yY" then
  main
 else
  print
  print "i don't understand."
  go'again
 endif
endproc go'again
//
proc timer
 for time'loop:=1 to 3000 do null
endproc timer
//
proc end'program
 print chr$(147)
 print "consult the crystal ball again"
 print "soon. have a nice day."
 end  // of program
endproc end'program
//
proc make'circle(radius,x0,y0) closed
 // draw circle of given radius
 //   centered at x0, y0
 // j. michener algorithm
 x:=0; y:=radius
 d:=3-2*radius
 while x<y do
  plot'sym'points(x,y,x0,y0)
  if d<0 then
   d:=d+4*x+6
  else
   d:=d+4*(x-y)+10
   y:=y-1
  endif
  x:=x+1
 endwhile
 if x=y then plot'sym'points(x,y,x0,y0)
endproc make'circle
//
proc plot'sym'points(x,y,x0,y0) closed
 xx:=x; yy:=y
 xy:=y; yx:=x
 adjust(x0,y0)
 adjust(xx,yy)
 adjust(xy,yx)
 plot x0+xx,y0+yy
 plot x0+xy,y0+yx
 plot x0+xx,y0-yy
 plot x0+xy,y0-yx
 plot x0-xx,y0-yy
 plot x0-xy,y0-yx
 plot x0-xx,y0+yy
 plot x0-xy,y0+yx
endproc plot'sym'points
//
proc adjust(ref x,ref y) closed
 scrunch:=1.34
 // note: scrunch corrects the difference
 //   in vertical and horizontal units
 x:=scrunch*x
endproc adjust ■
```

COMAL Kernal - 1985

Introduction

Superfluous or 1-productions have no semantics defined for them. Productions which just consist of a number of choices likewise have no semantics defined for them. We can assume that the semantics of A, where:

A ::= B | C | D

is the semantics of B, C, or D, and that we have a parser which can make the correct choice.

Conventions

::= "is defined as"
[ ] Optional
« » Metalinguistic Brackets
{ } May occur zero or more times
| Mutually exclusive alternatives

Note that the grammar is given mainly in iterative form, with the minimum of recursive definition.

Throughout this document "keywords" are shown in **UPPER CASE**.

Keywords are **RESERVED** and may **NOT** be used as identifiers. (A full list of reserved words is given at the end of this Kernal)

Note

This standard may be further revised. Prospective implementors of COMAL80 are advised to contact the group about any points not adequately covered by this document.

COMAL80 Standardisation Group,
Department of Computer & Information Science,
Linkoping University,
S-58183 Linkoping, Sweden
phone: +46-13-281000

Program Structure

«comal program» ::= «block»

«block» ::= {«declaration statement» |
«non declaration statement»}

The semantics of a block is the semantics of each of its component statements, taken in the context left over by executing the previous statements.

«declaration statement» ::=
«structured declaration statement» |
«unstructured declaration statement»

«non declaration statement» ::=
«structured statement» |
«unstructured statement»

«structured declaration statement» ::=
«procedure declaration» |
«function declaration»

«unstructured declaration statement» ::=
«dim statement» | «data statement»

«structured statement» ::=
«repetitive statement» |
«conditional statement»

The Structured Statements are composed of multiple lines. These lines are component lines. Each component line stands on its own as an input line and some syntax errors can be found. However further errors may be found when structures are checked, which can only be done when the program is complete.

«unstructured statement» ::=
«simple statement» «eol» |
«label statement» «eol» |
«eol»

«eol» ::= [«remark»] «newline»

more»

«remark» ::= //{«displayable character»}

A remark consists of a [possible empty] sequence of displayable characters, preceded by the sequence "//". The semantics of a remark is just its text with no interpretation by the system.

A comment can occur before any newline and will have no semantic effect on the preceding statements in the same line.

«newline» ::= implementation dependent

A newline is usually indicated by a carriage return, which is ASCII CHR$(13).

«displayable character» ::=
    implementation dependent

Displayable characters should include all displayable ASCII characters with values from 32 through 126.

«simple statement» ::= «return statement» |
    «stop statement» | «assignment statement» |
    «input statement» | «goto statement» |
    «restore statement» | «select statement» |
    «open statement» | «read statement» |
    «write statement» | «close statement» |
    «delete statement» | «print statement» |
    «zone statement» | «print using statement»|
    «procedure call statement» |
    «randomize statement»

Structured Statements

«repetitive statement» ::= «while statement» |
    «repeat statement» | «for statement»

«conditional statement» ::=
    «if statement» | «case statement»

«while statement» ::=
    WHILE «logical expression» DO «eol»
      «statement list»
    ENDWHILE «eol»

See the Extensions for the short one line while statement.

The expression is evaluated. If it is false (*zero*), control passes to the next statement after the ENDWHILE, and the only effect is any side effects generated by evaluating the expression. If the expression evaluates to true (*non-zero*), then the Statement list is executed. Control returns to the beginning of the WHILE statement, and the expression is re-evaluated. This pattern continues until the expression is found to be false. Control then passes to the statement following the ENDWHILE.

The statement list is indented when listed.

«statement list» ::=
    {«non declaration statement»}

The semantics of a sequence of non-declaration statements is the semantics of each component non-declaration statement in the context of the machine state which results from executing all the previous component non-declaration statements.

«repeat statement» ::=
    REPEAT «eol»
      «statement list»
    UNTIL «logical expression» «eol»

See the Extensions for the short one line repeat statement.

The statement list is executed. The expression is evaluated. If it is false (*zero*), control passes back to the first statement in the statement list. If the expression is true (*non-zero*), then control passes to the next statement following UNTIL.

more»

The statement list is indented when listed.

«for statement» ::= «short for statement» |
«long for statement»

«short for statement» ::=
FOR «for range» [«step»] DO
«simple statement» «eol»

«long for statement» ::=
FOR «for range» [«step»] DO «eol»
«statement list»
NEXT «control variable» «eol»

«for range» ::= «control variable» :=
«initial value» TO «final value»

«step» ::= STEP «step value»

«control variable» ::= «numeric identifier»

«initial value» ::= «numeric expression»

«final value» ::= «numeric expression»

«step value» ::= «numeric expression»

The For initialization is performed upon first entering the loop. This consists of evaluating the initial value and assigning the result to the control variable. Then the final value and the step value (if any) are evaluated. These three are evaluated once and for all and may not change during execution of the loop. The control variable is compared with the terminal value. In the case of a positive step value, if the control variable is less than or equal to the termination value, then the statement list is executed. In the case of a negative step value, it the control variable is greater then or equal to the termination value then the statement list is executed. In all other cases the statement list is not executed and control passes to the statement following the for statement. After each execution of the statement list, the value of the control variable is incremented by the step value. If the step part is null, then the control variable is incremented by one.

The statement list is indented when listed.

The semantics of the Short for statement is similar to the semantics of the long for statement, except that only a simple statement may occur after the DO.

If the control variable exceeds the final value on the first comparison, the statement list is not executed at all.

«if statement» ::=
«short if statement» | «long if statement»

«short if statement» ::=
IF «logical expression» THEN
«simple statement» «eol»

«long if statement» ::=
IF «logical expression» THEN «eol»
«statement list»
{ELIF «logical expression» THEN «eol»
«statement list»}
[ELSE «eol»
«statement list»]
ENDIF «eol»

«logical expression» ::= «numeric expression»

First the logical expression immediately following the IF is evaluated. If the value of the expression is true (non-zero), the statement list immediately following the THEN is executed, after which control passes to the statement following the ENDIF.

If the value of this expression is false (zero), and there are ELIF sections, the logical expressions of each of the ELIF statements are evaluated in order. As soon as one of these evaluates to true, the corresponding statement list is executed, and control then passes to the statement following the ENDIF.

more»

If none of the logical expressions *are* true, and there is an ELSE statement present, the statement list following the ELSE is executed, after which control passes to the statement after the ENDIF.

If none of the logical expressions *are* true, and there is no ELSE statement, control is passed to the statement after the ENDIF without any of the statement lists being executed.

The statement list is indented when listed.

The Short If statement behaves in the same way as the Long If statement, except that only a simple statement may occur after the THEN.

```
«case statement» ::=
    CASE «case selector» OF «eol»
    WHEN «choice list» «eol»
      «statement list»
    {WHEN «choice list» «eol»
      «statement list»}
    [OTHERWISE «eol»
      «statement list»]
    ENDCASE «eol»

«case selector» ::= «expression»

«choice list» ::= «numeric expression»
    {,«numeric expression»} |
    «string expression» {,«string expression»}
```

The case selector is evaluated. The choice lists are evaluated in the order in which they appear, until one is found with the same value as the case selector. When an identical value is found, then the statement list following the corresponding WHEN is executed. Control then passes to the statement following the ENDCASE. If none of the expressions in the WHEN choice lists match the CASE expression, the statement list following the OTHERWISE (if present) is executed. Control then passes to *the statement following the ENDCASE.*

Any Statements between the CASE .. OF line and first WHEN statement are ignored.

The statement list is indented when listed.

If none of the WHEN choice lists match the CASE selector, and no OTHERWISE clause exists, a runtime error occurs.

Structured Declarations

```
«procedure declaration» ::=
    PROC «procedure identifier»
    «head appendix» «eol»
      «procedure block»
    ENDPROC «procedure identifier» «eol»

«function declaration» ::=
    FUNC «function identifier»
    «head appendix» «eol»
      «function block»
    ENDFUNC «function identifier» «eol»

«function block» ::= «procedure block»

«return statement» ::= RETURN [«expression»]
```

Return causes the current procedure or function to be exited. A function **must** be exited through a return with an expression. A return with an expression is not allowed within a procedure. *Rather, a return without an expression may be used, but is not required.* A return of any sort may NOT be used within the main program.

```
«procedure block» ::=
    {«import statement»}
    {«unstructured declaration statement» |
    «non declaration statement»}

«head appendix» ::=
    [(«formal parameter list»)] [CLOSED]
```

more»

«procedure identifier» ::= «identifier»

«function identifier» ::=
    «numeric identifier» | «string identifier»

See the Extensions for the integer identifier.

«formal parameter list» ::=
    «formal parameter» {,«formal parameter»}

«formal parameter» ::=
    [REF] «variable identifier» |
    REF «variable identifier» «array indicator»

The Extensions also allow the REF to be optional for arrays.

«import statement» ::=
    IMPORT «import identifier»
    {,«import identifier»} «eol»

«import identifier» ::= «variable identifier» |
    «procedure identifier» | «function identifier»

«variable identifier» ::=
    «numeric identifier» | «string identifier»

«array indicator» ::= [({,})]

Functions and Procedures differ only in the way they are invoked and in the way they return values. A procedure is called by a procedure call, whereas a function is called by the use of the function identifier in an expression. The effect of a function invocation is to provide a single returned value of the same type as the function identifier (i.e. string or number). Procedures cannot return a value so procedure identifiers have no type (see also the RETURN statement).

The procedure or function can only be entered by calling it, and can only be exited by executing a RETURN statement or by control reaching the end of the procedure. The procedure or function identifier in the ENDPROC or ENDFUNC must match that in the heading. Parameters, other than arrays, are passed by value unless REF (pass by reference) is specified. Arrays are always passed by reference and REF must be included for them (*the Extensions allow REF to be optional for arrays too*). Their number of dimensions are indicated by commas. *No comma means one dimension, one comma means two dimensions, two commas mean three dimensions, etc.*

Procedures or functions are **Open** by default. **Closed** procedures or functions have their non-parameter variables allocated upon entry and these are lost when the procedure or function is exited.

The procedure block and function block are indented when listed.

Procedures or functions can be called recursively.

Unstructured Declarations

«dim statement» ::=
    DIM «declaration» {,«declaration»} «eol»

«declaration» ::=
    «numeric declaration» | «string declaration»

«numeric declaration» ::=
    «numeric identifier» («dimension part»)

«string declaration» ::=
    «string identifier» [(«dimension part»)]
    OF «length»

«dimension part» ::= «range» {,«range»}

«range» ::= [«lower bound»:] «upper bound»

«lower bound» ::= «numeric expression»

«upper bound» ::= «numeric expression»

more»

«length» ::= «numeric expression»

There can be arbitrary numbers of dimensions. If no lower bound is specified a default of 1 is assumed. If the upper bound is less than the lower bound, an error message will be issued. Each element of the items declared in the DIM statement is initialized. Numeric items are set to 0, and string items to the empty string. If an attempt is made to redimension an existing array a runtime error occurs. This means that one cannot make an array bigger by redimensioning it.

The Standards Group believes that dynamic string handling is preferable. If dynamic strings are implemented, the "OF «length»" part is optional. If the OF is present it sets the maximum length for the string. *The Extensions also allow the COMAL system to dimension a string to 40 characters maximum if the user does not specify otherwise.*

«data statement» ::=
    DATA «value» {,«value»} «eol»

All the DATA lists in the entire workspace are treated as one sequential "file" and are "consumed" by any READ statements that do NOT contain a file *designator.*

The Extensions allow data within CLOSED procedures and functions to be treated as local, not global to the whole program.

The execution of a RESTORE statement, without a label, causes the next READ operation to commence consuming input from the first DATA statement. If the RESTORE contains a label then the next READ is from the DATA statement following that label.

For good programming practice the data statement should only appear immediately before the end of the program or the end of a procedure.

«value» ::= [«sign»] «integer» |
    [«sign»] «real number» |
    «string constant» | TRUE | FALSE

«sign» ::= + | -

Because there is no boolean type the values TRUE and FALSE are in all respects equivalent to one and zero respectively.

False is always zero. True returns one, but when doing a comparison, any non-zero value is considered True.

Expressions

«expression» ::=
    «numeric expression» | «string expression»

«numeric expression» ::=
    [«numeric expression» OR] «logical term»

«logical term» ::=
    [«logical term» AND] «logical factor»

«logical factor» ::= [NOT] «relation»

«relation» ::=
    «string relation» | «arithmetic relation»

«string relation» ::= «string expression»
    «relational string operator»
    «string expression»

«relational string operator» ::=
    IN | «relational operator»

Syntax : «string1» IN «string2»

The two string expressions are evaluated. String2 is searched from the left until a copy of string1 is found. If no such copy is found the function returns FALSE (zero). Otherwise it returns the starting position of the first occurrence of string1. This must be non-zero and is therefore TRUE. If string1 is evaluated

more»

to the NULL string, then the value returned is 1. *NOTE: this is not how IN is implemented in some COMAL implementations, and is still under discussion. A proposal is currently submitted that specifies that the returned value should be false (zero).*

«arithmetic relation» ::=
    «formula» [«relational operator» «formula»]

«relational operator» ::=
    < | <= | = | >= | > | <>

«formula» ::= [«sign»] «arithmetic expression»

«arithmetic expression» ::=
    [«arithmetic expression» «adding operator»]
    «term»

«adding operator» ::= + | -

«term» ::=
    [«term» «multiplying operator»] «factor»

«multiplying operator» ::= * | / | DIV | MOD

The four *multiplying* operators have semantics as follows:

* and / perform the normal multiply and divide operations.

The DIV function is defined to take two arguments x and y and to return the next lowest integer less than or equal to the result of dividing x by y.

The MOD function also takes two arguments x and y (y must be positive) and returns (x - (x DIV y) * y).

These definitions of MOD and DIV are such as to give the following results:

```
   36 MOD 5 is 1      36 MOD (-5) is undefined
(-36) MOD 5 is 4   (-36) MOD (-5) is undefined
   35 MOD 5 is 0      35 MOD (-5) is undefined
(-35) MOD 5 is 0   (-35) MOD (-5) is undefined
   36 DIV 5 is 7      36 DIV (-5) is -8
(-36) DIV 5 is -8  (-36) DIV (-5) is 7
   35 DIV 5 is 7      35 DIV (-5) is -7
(-35) DIV 5 is -7  (-35) DIV (-5) is 7
```

Note: not all COMALs give these results for negative numbers.

«factor» ::= «operand» [^«factor»]

«operand» ::= («numeric expression») |
    «constant» | «numeric variable» |
    «numeric function call»

«constant» ::= «integer» | «real number» |
    TRUE | FALSE | *PI*

«real number» ::= «decimal number» [«exponent»]

«decimal number» ::= «integer»[.[«integer»]] |
    .«integer»

«exponent» ::= E [«sign»] «integer»

«integer» ::= «digit»{digit}

The allowable range of real and integer values is implementation dependent.

If an evaluation of a real expression results in underflow the value is set to zero. If the evaluation results in overflow, a run-time error occurs.

«numeric variable» ::=
    «numeric identifier» [(«subscript list»)]

«numeric identifier» ::= «real identifier»

«real identifier» ::= «identifier»

more»

If the string variable is given without a substring specifier then:

1. if the length of the string expression is greater than the length of the string variable, the string expression is truncated to the right.

2. if the string expression is shorter than or equal to the length of the string variable, the value is assigned and the actual length of the sting variable is set to the length of the string expression.

If the string variable refers to a substring then:

1. if the length of the string expression is greater than the length of the substring, the string expression is truncated to the right. If the string expression is shorter than or equal to the length of the substring, the value is assigned and the remaining part of the substring is space filled.

2. if the start index for the substring is greater than the actual length of the string variable + 1, a run-time error occurs.

Input Statement

«input statement» ::=
    INPUT [«string constant»:] «variable» «print end» |
    INPUT «file designator»: «variable list»

The Extensions also allow the string constant to be a string expression.

«variable list» ::= «variable» {,«variable»}

«variable» ::= «numeric variable» |
    «string variable»

«file designator» ::=
    FILE «channel number» [,«record number»]

«channel number» ::= «numeric expression»

«record number» ::= «numeric expression»

The «string constant» is used as a prompt. If the prompt is absent then a system standard prompt is supplied. The standard prompt is implementation dependent. (*We would like the standard default prompt to be declared. We suggest ? as the prompt.*) The system waits for the user to input a value. If the user makes a mistake and types in a value that does not match the type of the variable, the system prints an error message and re-issues the prompt. If «print end» is not specified the following print position will be the first position on the next line. If the «print end» is specified, the rules from the print statement apply.

If the file designator is present then no prompt is allowed as the input is read from an ASCII file. A file which is read by means of an INPUT statement is ASCII, whereas a file read by means of READ is binary.

Goto Statement

«goto statement» ::= GOTO «label identifier»

The GOTO statement causes transfer of control to the label statement with the corresponding label identifier. The GOTO is restricted in where it can jump to.

1. A GOTO cannot transfer control into a structured statement. One can have a label statement within a structured statement, but only statements inside the structured statement may GOTO it.

2. A GOTO cannot transfer control into or out of a procedure.

more»

«subscript list» ::= «subscript» {,«subscript»}
«subscript» ::= «numeric expression»

Rounding: If a real value is used as a subscript it is first rounded to the nearest integer value using the implicit rounding function which is defined to be (*the same as for the round statement in the Extensions*):

implicit_rounding_function(x) == int(x+0.5)

«numeric function call» ::=
    «numeric identifier»
    [(«actual parameter list»)]

«string expression» ::=
    «string operand» {+ «string operand»}

«string operand» ::= «string constant» |
    «string variable» | «string function call»

«string constant» ::= "{«displayable character»}"

«string variable» ::= «string identifier»
    [(«subscript list»)] [(«substring specifier»)]

«string identifier» ::= «identifier»$

«substring specifier» ::= «from»:«to»

The Extensions allow: [«from»:]«to»
If the from is omitted, only the character specified by the to is used.

«from» ::= «numeric expression»

«to» ::= «numeric expression»

«string function call» ::=
    «string identifier» [(«actual parameter list»)]
    [(«substring specifier»)]

«stop statement» ::= STOP

The Extensions allow a message after the STOP.

The Program is suspended and control returns to the COMAL system. All variables retain their current values, and the program may later be restarted at the statement immediately following the STOP.

«assignment statement» ::=
    «assignment» {;«assignment»}

«assignment» ::= «numeric assignment» |
    «string assignment»

«numeric assignment» ::=
    «numeric variable» := «numeric expression»

«string assignment» ::=
    «string variable» := «string expression»

In an expression containing both real and integer values the integers are "promoted" to reals and the expression evaluates to a real.

Precedence of Operators:

The precedence of the various operators in COMAL is shown in the following table. 1 is the highest precedence. Operators of the same precedence are evaluated left to right, except for exponentiation which is evaluated from right to left.

1) Exponentiation	^
2) Multiplying Operators	* | / | DIV | MOD
3) Adding Operators (Including Unary Minus)	+ | -
4) Relational Operators	= | <> | < | > | <= | >= | IN
5) Logical Negation	NOT
6) Logical ANDing	AND
7) Logical ORing	OR

If an operand is an expression within parentheses, the value to the operand is the value of that bracketed expression. It follows that the order of evaluation can be modified by inserting matching pairs of parentheses.

more»

3. Subject to 1. and 2. above, a GOTO may transfer control out of a structured statement.

«restore statement» ::=
RESTORE [«label identifier»]

The pointer to the next data item is reset to point to the first item in the (textually) first DATA statement. If the label is included the pointer is restored only to the DATA statement immediately following that label.

«label statement» ::= «label identifier»:

«label identifier» ::= «identifier»

The Label statement is included as an unstructured statement. It may appear within a structured statement, although it only has scope (i.e. can only be jumped to) within the structured statement. Note that one may not include a label statement on the same line as another statement.

A label statement is also used to separate data statements (see the RESTORE statement).

File Handling

«select statement» ::=
SELECT «type» «device specifier»

[,«dev info»] had been included after «device specifier». It was used in an older version of the Kernal, but since then «dev info» was deleted from the Kernal and all references to it also should be deleted.

«type» ::= OUTPUT

The Extensions should also allow INPUT as a type.

«device specifier» ::= «string expression»

This string expression can be a file name or peripheral device identification (such as a printer).

«open statement» ::=
OPEN FILE «channel number»,«file name»,
«mode»

«file name» ::= «string expression»

The file name is assumed to exist on the default disk drive, unless another drive or device is specified. To specify a specific drive, precede the actual file name with [«drive id»:] (the colon separates the id from the file name). Computer systems use various methods of identifying a disk drive. However, the COMAL system can accept one method, and then (transparent to the user) convert it into the method required by the computer system. This has not been decided yet.

«mode» ::= READ | WRITE | APPEND |
RANDOM «record length» [«random mode»]

«random mode» ::= READONLY | WRITEONLY

«record length» ::= «numeric expression»

If mode is random all records have the specified length. The data, if any, in the record need not fill the record. The string expression should evaluate to a file name. If the file does or does not exist, the system will respond according to the following plan:

mode	yes	no
read	open	run-time error
write	run-time error	create
append	open	create
random	open	create
random readonly	open	run-time error
random writeonly	open	create

If no error has occurred, the file will be associated with the channel number given.

more»

«read statement» ::= READ «variable list» |
    READ «file designator»: «variable list»

If the variable type in the read statement is incompatible with the type of the value in the data statement, a run-time error occurs.

«write statement» ::=
    WRITE «file designator»: «variable list»

A record number in the file designator is valid only if the file is opened in random access mode. Otherwise the presence of a record number causes an error to occur. The expressions are written out in binary (if they are numeric) or in ASCII (if they are text) to the file opened on the given channel. If no file is attached to the channel, an error results. If the file is random access, then the combined bulk of all the data in the expression list should be less than or equal to the record length.

«delete statement» ::=
    DELETE «file name»

If the named file is currently open, a runtime error occurs. If it is closed, it is deleted, and if it is reopened, it will be a new file. It is permitted to delete a non-existing file, and no error results.

«close statement» ::=
    CLOSE [FILE «channel number»]

The file attached to the channel specified by channel number is disconnected. If the channel number given is not attached to any file, an error occurs. If no channel number is specified than ALL currently open files are closed. *If there are no open files, no error results.*

Print Statement

«print statement» ::= PRINT «output list» |
    PRINT «file designator»: «output list»

«output list» ::= [«print list» [«print end»]]

«print list» ::= «print element»
    {«print separator» «print element»}

«print element» ::= «expression» | «tab function»

«print end» ::= «print separator»

«print separator» ::= , | ;

We also are suggesting the ! or : as a null separator for no spaces between items.

«tab function» ::= TAB(«numeric expression»)

TAB causes the next printed item to go in the column position given by the expression. The first position allowed is defined to be 1. A TAB with a non-positive argument gives a runtime error. The maximum argument permitted is implementation dependent. If the argument is less than the current cursor position, it has no effect.

If a file designator is specified, then the ASCII output goes out to the file rather than to the user's terminal. Formatted information may be written out to ASCII files.

Normally the expressions are printed out using a system-defined format for the particular type of expression.

The print line is divided into print zones. The zone width is determined by the zone statement. The default zone width is 0. An odd sized zone may occur at the end of the print line. Printing zone divided lines may be compared to using the tabulator key on an ordinary typewriter. This means that after a print element is output the tabulator key is pressed. As a consequence of this the next print element starts in the leftmost position of the next zone, and that printing a print element of the same length as the zone width

more»

leaves the following zone space filled. Zone width 0 means that no spacing between print elements occurs.

A semicolon used as print separator is output as one space. *We propose the exclamation point or colon as a null print separator, always outputing nothing between items.* The effect of a «print end» is the same as of a «print separator». If no «print end» is specified, printing is continued in the first position on the next line. A print statement with no «print list» causes output of a newline.

«print using statement» ::=
    PRINT USING «format info»: «using list» [«print end»] |
    PRINT «file designator»: USING «format info»: «using list» [«print end»]

«using list» ::=
    «using element» {,«using element»}

«using element» ::= «numeric expression»

«format info» ::= «string expression»

In the PRINT USING statement the format is determined from the string in the format clause. The meaning of the characters in the string of the Format Info is as follows: A substring of one or more embedded hash signs (#), optionally containing a single point (.), will be substituted on output by the value of a corresponding numeric expression. The expression will be printed out with as many digits precision as specified by hash signs to the right of the point. If no point is specified, neither the decimal point nor any fractional part is printed. Leading zeroes in the integer part are replaced by *spaces*; trailing zeroes in the fractional part are printed. If the expression has an integer part which is too large to be represented in the field, the field is printed out as all *asterisks* "*".

Procedure Call

«procedure call statement» ::=
    [EXEC] «procedure identifier» [(«actual parameter list»)]

Syntactically, the only restriction on actual parameters is that they must be expressions. However, in order to send results back via a parameter passed by reference, the actual parameter must have a l-value. (e.g. the l-value of X is the address X, but the expression X+Y has no l-value. *A REF parameter can only pass results to a variable, not a constant or expression.*)

A procedure may have side effects, both **through parameters** passed by reference and through shared variables.

SCOPE : Within a given scope, the first use of a name, whether as a procedure name, function name, label or variable name (whether DIMmed or not) defines the names type. Any subsequent attempt to redefine this name is an error.

Some systems do allow a name to be used in more than one way: test$ and test#.

BINDING :
1. Procedure and function names must be IMPORTed into CLOSED procedures.
2. OPEN procedures or functions canNOT be IMPORTed. *Some systems do allow OPEN procedures to be IMPORTed.*

Advisory Note

To ensure portability of software, users procedures should NOT rely on their formal parameters being available to called procedures.

«actual parameter list» ::=
    «actual parameter» {,«actual parameter»}

«actual parameter» ::= «expression»

more»

«zone statement» ::= ZONE «numeric expression»

ZONE sets the default spacing for items output with the PRINT statement.

The Extensions allow ZONE to be used as a function that returns the current zone setting.

«randomize statement» ::=
RANDOMIZE [«numeric expression»]

Prior to the execution of any program the random number generator is seeded by a random value.

The execution of a randomize statement without a numeric expression means that the random number generator is seeded by a random value.

The execution of a randomize statement with a numeric expression means that the random number generator is seeded by the value of the numeric expression.

«identifier» ::= «letter» {«letter» | «digit» | _}

The apostrophe ' may also be used as part of an identifier, just like the underscore.

If underscore is not available on the terminal to be used, another character may be chosen.

«letter» ::= implementation dependent

A letter must include all upper and lower case ASCII letters. It may optionally include other national characters.

«digit» ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Standard Built-in Functions

COS(«numeric expression»)
SIN(«numeric expression»)
TAN(«numeric expression»)
ATN(«numeric expression»)

-- All the trigonometric function take a numeric argument whose value is assumed to be in radians. The value returned is real. An invalid argument causes a runtime error.

Some systems allow the user to choose to use degrees rather than radians.

ABS(«numeric expression»)

-- Returns the **absolute** value of the expression.

LOG(«numeric expression»)
EXP(«numeric expression»)

-- The LOG and EXP functions find the natural log and exponent values of a REAL argument.

SQR(«numeric expression»)

-- *SQR requires that the numeric expression be positive. Non-positive values result in a run time error.*

INT(«numeric expression»)

-- returns the next smallest integer value. Thus INT(-3.5) returns -4.

SGN(«numeric expression»)

-- returns -1 for a negative number, 0 for 0 and +1 for a positive number.

RND

-- returns a "random" value greater than or equal to zero, and less than one.

RND(«numeric expression»,«numeric expression»)

-- returns a "random" *integer* in the range given (*inclusive*).

more»

LEN(«string expression»)

-- returns the *current* length of a string

ORD(«string expression»)

-- returns the internal representation (*ASCII*) of the first character of its argument. *The null string results in a value of -1 (proposed).*

VAL(«string expression»)

-- returns the numeric equivalent of its string expression argument. The argument can have the format:

[«sign»]«integer»[.«integer»][E«integer»]

or

[«sign»].«integer»[E«integer»]

No blanks are permitted in the string.

STR$(«numeric expression»)

-- PRINT x and PRINT STR$(x) always produce the same result.

CHR$(«numeric expression»)

-- CHR$ takes an internal representation (*ASCII*) and returns the equivalent character. It is the inverse of ORD.

EOF(«numeric expression»)

-- the numeric expression gives a channel number. EOF becomes true when there are no more records to be read. EOF of a closed file causes a runtime error. EOF becomes true if an empty file is opened.

EOD

-- EOD becomes true in the same way *as EOF* except that the file is the internal data file.

STANDARD EXTENSIONS

Integer Type Variables

To allow variables of type integer, replace the production:

«numeric identifier» ::= «real identifier»

with the two productions:

«numeric identifier» ::= «real identifier» |
    «integer identifier»

«integer identifier» ::= «identifier»#

Some flexibility between real and integers is allowed. If an integer expression is assigned to a real variable an implicit FLOAT() takes place without any errors. If a real expression is assigned to an integer variable, an implicit FIX() takes place. Two functions for rounding are provided. These are ROUND() and ROUNDEVEN().

Short WHILE and REPEAT

One line versions of these statements are provided by modifying the productions for the Repetitive Statements.

«while statement» ::= «short while statement» |
    «long while statement»

«short while statement» ::= WHILE
    «logical expression» DO «simple statement»
    «eol»

«long while statement» ::=
    WHILE «logical expression» DO «eol»
      «statement list»
    ENDWHILE «eol»

«repeat statement» ::= «short repeat statement»|
    «long repeat statement»

more»

«short repeat statement» ::= REPEAT
«simple statement» UNTIL
«logical expression» «eol»

«long repeat statement» ::=
REPEAT *«eol»*
«statement list»
UNTIL «logical expression» «eol»

STOP With a Message

Instead of printing line numbers a STOP statement can result in a message being output. The syntax is:

«stop statement» ::= STOP[«string expression»]

Static Strings

If dynamic strings are NOT implemented a default string-length of **40** (forty) is given to an unDIMmed string variable when it is first encountered.

Parameter Lists to Procedures

To drop the requirement that the parameter list to a procedure be parenthesised, replace the production

«procedure call statement» ::=
[EXEC] «procedure identifier»
[(«actual parameter list»)]

with the following productions.

«procedure call statement» ::= [EXEC]
«procedure identifier»[(«parameter list»)]
«parameter list» ::=
«parenthesised parameter list» |
«actual parameter list»

«parenthesised parameter list» ::=
(«actual parameter list»)

Add to the built in functions:

«zone function» ::= ZONE

This returns the current zone setting.

«rounding function» ::=
ROUND(«numeric expression»)

This rounds a value to the nearest integer.

Add to the simple statement:

«clear screen statement» ::= PAGE

Reserved Words in COMAL 80

ABS	FLOAT	REF
AND	FOR	REPEAT
APPEND	FUNC	RESTORE
ATN	GOTO	RETURN
CASE	IF	RND
CHR$	IMPORT	ROUND
COS	IN	ROUNDEVEN
CLOSE	INPUT	SGN
CLOSED	INT	SIN
DATA	LEN	SQR
DIM	LOG	STEP
DIV	MOD	STOP
DO	NEXT	STR$
ELIF	NOT	TAB
ELSE	OF	TAN
ENDCASE	ON	THEN
ENDFUNC	OPEN	TO
ENDPROC	OR	TRUE
ENDWHILE	ORD	UNTIL
EOD	OTHERWISE	USING
EOF	PRINT	VAL
EXEC	PROC	WHEN
EXP	RANDOM	WHILE
FALSE	RANDOMIZE	WRITE
FILE	READ	WRITEONLY
FIX	READONLY	*PAGE*

[*This is the April 1985 COMAL Kernal, the latest edition that we have. Items in italics are our own clarifications and comments, which we are proposing to be added into the Kernal.*] ■

How to Use the Kernal

by Richard Bain

The last several pages of this newsletter have been dedicated to listing the COMAL Kernal. This article and program help explain how to understand and use the Kernal. The program parallels a portion of the COMAL Kernal. The program is also on *Today Disk #17*.

The COMAL Kernal is not easy to understand until you have had training in reading the modified Backus-Naur Form (BNF) used in its definitions. The following program paraphrases the Kernal definition of an «expression». It goes one step further though. The COMAL Kernal can be used to formally verify that a program syntactically meets the COMAL standard. However, it does not quite tell what a program will do when it executes. The comments (semantics) below each definition explain what the program will do.

The program below allows the user to type in an «expression». It has one procedure for each definition used from the Kernal. Each procedure insures that the correct syntactic element is present. It then appends a command to a string organized in Reverse Polish Notation (RPN). The command may be a number or an operator. Finally, the RPN expression is evaluated.

Note: RPN is a system commonly used in Hewlett Packard calculators. It places the operands (numbers) before the operators. When the expression is evaluated, operands (numbers) are placed on a stack. When an operand (+ * etc.) is encountered, two numbers are popped off the stack, the operation is evaluated, and the result is pushed back on the stack. If all goes well, there will be exactly one number on the stack to indicate the result of the expression. For example: the expression 1+2 would be converted to 1 2 + in RPN. To evaluate it, first push the 1 and 2 on the stack. The + will cause the numbers to be popped of the stack (the stack is now empty). The numbers are added together and the result (3) is pushed back on the stack. 3 is the result of the expression.

The main program prints the instructions and then enters a loop to let the user enter the «expression». If the «expression» is valid, it is evaluated. Procedure expression directs its sub-procedures and insures that the expression is completely evaluated. One *trick* it uses is to add three characters to the string containing the expression to be evaluated. A copy of the original string must be must be used because the original may not be dimensioned large enough to hold the additional characters.

The first character added to the «expression» string is **CHR$(13)** to mark the end of line. It is easier to test if a character represents the end of line than to test if a character exists. The second character is an error flag. It is initialized to **CHR$(0)** for no error, but will be changed to **CHR$(1)** if an error occurs. The last character appended to the string is a pointer to the current position within the expression. It initially points to the first character in the string. After the expression is evaluated, it must be pointing to the end of line character, or an error has occurred.

The notation used in the rest of this article is defined in the COMAL Kernal article on page 40. Briefly, any name inside «» represents a syntactical unit in COMAL. It must be defined precisely later. Any item not enclosed in «» must be entered exactly as it appears, except uppercase letters may be typed in lowercase. The symbol ::= means that the name on the left is defined to be the item on the right. If an item to the right of ::= is inside [], that item is

more»

optional, it may or may not be used as part the definition for the name on the left. If two or more items to the right of ::= are separated by |, the name is being defined as one or the other of the items, but not both. This will be clarified later.

«expression»::=«numeric expression» |
«string expression»

An «expression» must either be a «numeric expression» or a «string expression». Procedure **expression** calls procedure **numeric'expression** to start breaking down the «expression» into manageable chunks. String expressions such as **LEN(a$)** are not implemented in this program.

«numeric expression»::=
[«numeric expression» OR] «logical term»

The Kernal definition of a «numeric expression» contains an optional part ([«numeric expression» OR]) and a mandatory part («logical term»). However, the **numeric'expression** procedure cannot recursively call itself to see if the option part is there. (This could go on forever.) Fortunately, there is another option. The procedure can first get the «logical term». Whatever a «logical term» is, it must by definition also be a «numeric expression». The following item, if any, must be the operator **OR**. (Anything else would indicated the end of the «numeric expression».)

If **OR** is present, we have found the optional part of the «numeric expression», namely [«numeric expression» OR]. We must proceed to get the mandatory part, the «logical term». After this, **OR** is added to the RPN expression. (Remember, operators such as **OR** go after their operands in an RPN expression.) **OR** acts on the «numeric expression» and the «logical term» which were already placed in the RPN expression (see below). Note: once we find the **OR** and the «logical term», we again have a «numeric expression». It may also turn out to be followed by **OR** and another «logical term» and so on. This is why **OR** is checked for in a **WHILE** loop rather than an **IF** statement.

«logical term»::=[«logical term» AND]
«logical factor»

Procedure **numeric'expression** needed a «logical term». Procedure **logical'term** provides it. A «logical term» has an optional [«logical term» AND] followed by a mandatory «logical factor». The above arguments can be repeated exactly, just use the words *logical term* for *numeric expression*, *and* for *or*, and *logical factor* for *logical term*. Thus we first get the «logical factor». A **WHILE** loop is used to check if **AND** is the next item in the «expression». If it is, we have found an optional [«logical term» AND]. We then fetch the mandatory «logical factor» and so on.

«logical factor»::=[NOT]«relation»
«relation»::=«string relation» |
«arithmetic relation»

A «logical factor» has a different type of definition than the above examples. It has an optional [NOT] followed by a mandatory «relation». A «relation» can be a «string relation» or an «arithmetic relation», but this program only deals with the latter. This definition is not recursive («logical factor» is not part of the definition for a «logical factor»). Therefore, an **IF** statement is used in this example instead of a **WHILE** loop. Note, if **NOT** is included as part of the «logical factor», it is placed in the RPN expression **after** the «arithmetic relation».

«arithmetic relation»::=«formula»
[«relational operator»«formula»]

An «arithmetic relation» is a «formula» possibly followed by a «relational operator» and another

more»

«formula». Again, this definition is not recursive. This prevents an «arithmetic relation» such as **4>3>2**. This makes some sense. After all, **4>3** is **TRUE** which evaluates to **1**. **1>2** is **FALSE** so the above expression would be **FALSE**. Maybe it is just as well that it isn't allowed.

«formula»::=[«sign»]«arithmetic expression»

A «formula» is an optional [«sign»] followed by an «arithmetic expression». An «arithmetic expression» is something like **2+3** (we'll get to that soon). By allowing the optional [«sign»] (+ or -) in this part of the BNF, a valid «formula» could be **-2+3**, but not **2+-3**. Try this on your computer and see for yourself. This definition of a «formula» is syntactically pleasing, but complicates the evaluation of an «arithmetic expression».

«arithmetic expression»::=
    [«arithmetic expression»«adding operator»]
    «term»

An «arithmetic expression» is an optional [«arithmetic expression»«adding operator»] followed by a mandatory «term». This syntax is similar to that of the first two examples, «numeric expression» and «logical term». The structure of the procedure is the same except for one thing. After obtaining the first «term», there is test with the variable **negate**. This variable is set to **TRUE** if the «formula» above started with a unary minus sign. Normally, an operand such as the unary minus will apply to the entire syntactical structure next to it (the «arithmetic expression»). However, precedence rules do not allow this. A close look at the precedence rules in combination with the BNF of the Kernal indicate that the unary minus sign only applies to the **first** «term» of the «arithmetic expression». (**-2+3** is **(-2)+3**, not **-(2+3)**.) Hence the existence of this sign must be passed as a flag (parameter) to procedure **arithmetic'expression** to allow the sign to be placed correctly in the RPN expression. Note: *-n-* was used for the unary minus to distinguish it from the diadic minus used in subtraction.

«term»::=[«term»«multiplying operator»]«factor»

A «term» is an optional [«term»«multiplying operator»] followed by a mandatory «factor». By now, you should be able to clearly understand this from the program listing.

«factor»::=«operand»[^«factor»]

A «factor» is an «operand» possibly followed by [^«factor»]. Note: this definition is **right** associative; this is clearly explained in the semantics portion of the COMAL Kernal in the operator precedence section. This means that **2^3^4** is evaluated as **2^(3^4)** instead of **(2^3)^4**. Procedure **factor** recursively calls itself from within a **WHILE** loop if the optional ^ is present. If the definition of «factor» were left associative instead of right associative, the recursive call to **factor** would be replaced with another call to **operand**.

«operand»::=(«numeric expression»)|«constant»|
    «numeric variable»|«numeric function call»

So far every definition has depended on some other definition. You may have followed 8 procedures to see if **13** is a valid «expression» and still don't have a conclusive answer. «Operand» is the common thread you have been looking for. It is not the end of the chain, but it is the link that actually returns a number. Procedure **operand** places a number in the RPN expression so the operators mentioned above will have operands to work on. The number may be in the form of a constant, a function call, or even another «numeric expression» contained within parentheses. Note: it **cannot** be an «expression» contained within parentheses. This makes it possible for procedure **expression** to

more»

check for end of line, but procedure numeric'expression doesn't care if the line has ended or not. I leave it for the reader to verify that 13 is truly an «operand». *Hint, 13 is a «constant». See the Kernal for details.*

The intent of this program was to show that the COMAL Kernal has meaning, and that a programmer can make use of it. The concepts presented here are the only ones you need to understand the entire Kernal. However, I realize that some people out there are more interested in running a program than learning from it. For these people, I have included a procedure to evaluate the RPN expression. There is nothing complicated about it (most operations require one or two pops and a push) and I see no need to go into further detail.

```
dim algebra$ of 80, rpn$ of 256, reply$ of 1
instructions
repeat
  page
  print "type a function:"
  print
  input "function: ": algebra$
  if len(algebra$)>0 then
    expression(algebra$,rpn$)
    if len(rpn$)>0 then
      print
      print "reverse polish notation: "
      print rpn$
      print
      print "result: ",eval(rpn$)
    endif
    print
    input "press <return> to continue (q to
    quit)": reply$ // wrap line
  endif
until reply$="Q" or reply$="q"
//
proc instructions
  page
  print at 3,5: "this program allows the user to"
  print "enter a numeric function. the function"
  print "is checked for proper syntax and is"
  print "translated to reverse polish notation."
  print "if the original function was accepted,"
  print "the rpn function will be evaluated."
  print "note: the rpn expression isn't checked"
  print "for problems such as division by zero."
  print
  print "constants: true false pi"
  print
  print "valid operators: + - * / mod div ^"
  print "      and or not < > = <= >= <>"
  print
  print "functions: sin(x) cos(x) tan(x) atn(x)"
  print "           log(x) exp(x) abs(x) int(x)"
  print "           sgn(x) sqr(x)"
  print
  input "press <return> to continue ": reply$
endproc instructions
//
proc expression(aos$,ref rpn$) closed
  // <expression>::=<string expression> !
  //                <numeric expression>
  // <string expression> is not implemented
  dim alg$ of len(aos$)+3, sym$ of len(aos$)
  alg$:=aos$+chr$(13)+chr$(0)+chr$(1)
  // chr$(13) is <end of line> marker
  // chr$(0) is error flag, error sets it to 1
  // chr$(1) points to next character in alg$
  rpn$:=""; sym$:=""
  numeric'expression(alg$,rpn$,sym$)
  if sym$<>chr$(13) then
    error(sym$+" not expected")
  endif
  if alg$(len(alg$)-1:len(alg$)-1)<>chr$(0)
  then rpn$:="" // wrap line
  //
  proc numeric'expression(ref alg$,ref rpn$,ref
    sym$) closed // wrap line
    // <numeric expression>::=
    //    [<numeric expression>or]<logical term>
    import logical'term
    import logical'factor,arithmetic'relation
    import formula,arithmetic'expression,term
    import factor,operand,error,get'sym$
    sym$:=get'sym$(alg$)
    logical'term
    while sym$="or" do
```

more»

```
    sym$:=get'sym$(alg$)
    logical'term
    rpn$:+"or "
  endwhile
endproc numeric'expression
//
proc logical'term closed
  // <logical'term>::=
  //    [<logical term>and]<logical factor>
  import numeric'expression
  import logical'factor,arithmetic'relation
  import formula,arithmetic'expression,term
  import factor,operand,error,get'sym$
  import alg$,rpn$,sym$
  logical'factor
  while sym$="and" do
    sym$:=get'sym$(alg$)
    logical'factor
    rpn$:+"and "
  endwhile
endproc logical'term
//
proc logical'factor closed
  // <logical'factor>::=[not]<relation>
  // <relation>::=<string relation> !
  //              <arithmetic relation>
  // <string relation> is not implemented
  import numeric'expression,logical'term
  import arithmetic'relation
  import formula,arithmetic'expression,term
  import factor,operand,error,get'sym$
  import alg$,rpn$,sym$
  if sym$="not" then
    sym$:=get'sym$(alg$)
    arithmetic'relation
    rpn$:+"not "
  else
    arithmetic'relation
  endif
endproc logical'factor
//
proc arithmetic'relation closed
  // <arithmetic'relation>::=
  //     <formula>[<relational operator>
  //     <formula>]
  // <relational operator>::=
  //     < ! <= ! = ! >= ! > ! <>
  import numeric'expression,logical'term
  import logical'factor
  import formula,arithmetic'expression,term
  import factor,operand,error,get'sym$
  import alg$,rpn$,sym$
  dim op$ of 2
  formula
  op$:=sym$
  if op$(1:1) in "<>=" then
    sym$:=get'sym$(alg$)
    formula
    rpn$:+op$+" "
  endif
endproc arithmetic'relation
//
proc formula closed
  // <formula>::=
  //    [<sign>]<arithmetic expression>
  // the <sign> might not apply to the
  // entire <arithmetic expression>
  // due to precedence rules
  import numeric'expression,logical'term
  import logical'factor,arithmetic'relation
  import arithmetic'expression,term
  import factor,operand,error,get'sym$
  import alg$,rpn$,sym$
  dim sign$ of 1
  sign$:=sym$
  if sign$ in "+-" then sym$:=get'sym$(alg$)
  arithmetic'expression(sign$="-")
endproc formula
//
proc arithmetic'expression(negate) closed
  // <arithmetic expression>::=
  //     [<arithmetic expression>
  //     <adding operator>]<term>
  // <adding operator>::= + ! -
  import numeric'expression,logical'term
  import logical'factor,arithmetic'relation
  import formula,term
  import factor,operand,error,get'sym$
  import alg$,rpn$,sym$
  dim op$ of 1
  term
  if negate then rpn$:+"-n- "
```

more»

```
  while sym$ in "+-" do
    op$:=sym$
    sym$:=get'sym$(alg$)
    term
    rpn$:+op$+" "
  endwhile
endproc arithmetic'expression
//
proc term closed
  // <term>::=[<term>
  //     <multiplying operator>]<factor>
  // <multiplying operator>::=
  //     * ! / ! Mod ! Div
  import numeric'expression,logical'term
  import logical'factor,arithmetic'relation
  import formula,arithmetic'expression
  import factor,operand,error,get'sym$
  import alg$,rpn$,sym$
  dim op$ of 4
  factor
  op$:=sym$
  while " "+op$+" " in " * / div mod " do
    sym$:=get'sym$(alg$)
    factor
    rpn$:+op$+" "
    op$:=sym$
  endwhile
endproc term
//
proc factor closed
  // <factor>::=<operand>[^<factor>]
  //     right associative (kernal)
  // <factor>::=[<factor>^]<operand>
  //     left associative (more correct?)
  import numeric'expression,logical'term
  import logical'factor,arithmetic'relation
  import formula,arithmetic'expression,term
  import operand,error,get'sym$
  import alg$,rpn$,sym$
  operand
  while sym$="^" do
    sym$:=get'sym$(alg$)
    factor
    rpn$:+"^ "
  endwhile
endproc factor
//
proc operand closed
  // <operand>::=
  //    (<numeric expression>) ! <constant> !
  //    <numeric variable> !
  //    <numeric function call>
  // <constant>::=<integer> ! <real number> !
  //    true ! false
  //    pi is treated like a constant
  // <numeric variable> is not implemented
  // <numeric function call> is restricted
  //    to system functions
  import numeric'expression,logical'term
  import logical'factor,arithmetic'relation
  import formula,arithmetic'expression,term
  import factor,error,get'sym$
  import alg$,rpn$,sym$
  dim var$ of len(alg$)
  if sym$="(" then
    parentheses
  elif sym$(1:1) in "1234567890." then
    rpn$:+sym$+" "
    sym$:=get'sym$(alg$)
  elif " "+sym$+" " in " pi true false " then
    rpn$:+sym$+" "
    sym$:=get'sym$(alg$)
  elif " "+sym$+" " in " sin cos tan atn log
    exp abs int sgn sqr " then // wrap line
    var$:=sym$
    sym$:=get'sym$(alg$)
    if sym$="(" then
      parentheses
    else
      error("( expected")
    endif
    rpn$:+var$+" "
  else
    error("operand expected")
  endif
  //
  proc parentheses
    numeric'expression(alg$,rpn$,sym$)
    if sym$=")" then
      sym$:=get'sym$(alg$)
    else
      error(") expected")
```

more»

```
    endif
  endproc parentheses
endproc operand
//
func get'sym$(ref alg$) closed
  import error
  dim temp$ of 80, char$ of 1, letters$ of 26
  dim numerals$ of 11
  letters$:="abcdefghijklmnopqrstuvwxyz"
  numerals$:="1234567890"
  remove'spaces(alg$)
  temp$:=next'char$
  if temp$ in numerals$+"." then
    temp$:=get'number$
  elif temp$="<" then
    remove'first
    if next'char$ in ">=" then
      temp$:+next'char$
      remove'first
    endif
  elif temp$=">" then
    remove'first
    if next'char$="=" then
      temp$:+"="
      remove'first
    endif
  elif temp$ in letters$ then
    remove'first
    char$:=next'char$
    while char$ in letters$+numerals$ do
      temp$:+char$
      remove'first
      char$:=next'char$
    endwhile
  elif temp$=chr$(13) then // <eol> do null
  else
    remove'first
  endif
  return temp$
  //
  proc remove'spaces(ref alg$) closed
    length:=len(alg$)
    now:=ord(alg$(length:length))
    while alg$(now:now)=" " do
      alg$(length:length):=chr$(ord(alg$(
      length:length))+1) // wrap line
      now:+1
    endwhile
  endproc remove'spaces
  //
  proc remove'first closed
    import alg$
    length:=len(alg$)
    alg$(length:length):=chr$(ord(alg$(
    length:length))+1) // wrap line
  endproc remove'first
  //
  func next'char$ closed
    import alg$
    dim c$ of 1
    now:=ord(alg$(len(alg$):len(alg$)))
    c$:=alg$(now:now)
    up:=ord(c$)-ord("a")
    if up>=0 and up<26 then c$:=chr$(
    ord("a")+up) // wrap line
    return c$
  endfunc next'char$
  //
  func get'number$ closed
    import next'char$,remove'first,error
    import alg$
    dim ch$ of 1, number$ of 80
    ch$:=next'char$; number$:=""
    if ch$="." then
      number$:+"."
      remove'first
      integer
    else
      integer
      if next'char$="." then
        number$:+"."
        remove'first
        if next'char$ in "1234567890"
        then integer // wrap line
      endif
    endif       if next'char$="e" then
      number$:+"e"
      remove'first
      if next'char$ in "+-" then
        number$:+next'char$
        remove'first
      endif
```

more»

```
        integer
      endif
      return number$
      //
      proc integer
        if next'char$ in "1234567890" then
          repeat
            number$:+next'char$
            remove'first
          until not next'char$ in "1234567890"
        else
          error("integer expected")
        endif
      endproc integer
    endfunc get'number$
  endfunc get'sym$
  //
  proc error(msg$) closed
    import alg$,sym$
    length:=len(alg$)
    if ord(alg$(length-1:length-1))=0 then
      current:=ord(alg$(length:length))
      tab'to:=current+9-len(sym$)+(sym$=
      chr$(13)) // wrap line
      for t:=1 to tab'to do print " ",
      print "^ "
      print msg$
      alg$(length-1:length):=chr$(1)+chr$(
      length-1) // wrap line
    endif
  endproc error
endproc expression
//
func eval(rpn$) closed
  dim stack(100), item$ of 10
  top:=0
  while len(rpn$) do
    temp:=" " in rpn$
    item$:=rpn$(1:temp-1)
    if temp=len(rpn$) then
      rpn$:=""
    else
      rpn$:=rpn$(temp+1:len(rpn$))
    endif
    if item$(1:1) in "1234567890." then
      push(val(item$))
    else
      case item$ of
      when "-n-"
        push(-pop)
      when "pi"
        push(pi)
      when "true"
        push(true)
      when "false"
        push(false)
      when "+"
        r:=pop; l:=pop
        push(l+r)
      when "-"
        r:=pop; l:=pop
        push(l-r)
      when "*"
        r:=pop; l:=pop
        push(l*r)
      when "/"
        r:=pop; l:=pop
        push(l/r)
      when "div"
        r:=pop; l:=pop
        push(l div r)
      when "mod"
        r:=pop; l:=pop
        push(l mod r)
      when "^"
        r:=pop; l:=pop
        push(l^r)
      when "and"
        r:=pop; l:=pop
        push(l and r)
      when "or"
        r:=pop; l:=pop
        push(l or r)
      when "not"
        push(not pop)
      when ">"
        r:=pop; l:=pop
        push(l>r)
      when "<"
        r:=pop; l:=pop
        push(l<r)
      when "="
        r:=pop; l:=pop
        push(l=r)
      when ">="
        r:=pop; l:=pop
        push(l>=r)
      when "<="
        r:=pop; l:=pop
        push(l<=r)
      when "<>"
        r:=pop; l:=pop
        push(l<>r)
      when "sin"
        push(sin(pop))
      when "cos"
        push(cos(pop))
      when "tan"
        push(tan(pop))
      when "atn"
        push(atn(pop))
      when "exp"
        push(exp(pop))
      when "log"
        push(log(pop))
      when "abs"
        push(abs(pop))
      when "int"
        push(int(pop))
      when "sgn"
        push(sgn(pop))
      when "sqr"
        push(sqr(pop))
      endcase
    endif
  endwhile
  return stack(1)
  //
  func pop
    top:-1
    return stack(top+1)
  endfunc pop
  //
  proc push(n)
    top:+1
    stack(top):=n
  endproc push
endfunc eval ■
```

C128 COMAL Review

The c128 COMAL cartridge adds some very nice features to COMAL on the c128. It can run in fast or slow mode. You can use either the 40 or 80 column screen with text windows for program editing and execution. You have 40K of program and variable storage. An additional 40K can be used for special RAM files. You have full access to the 1571 disk drive features, but the RAM expanders (ie, 1750) do not appear to be directly supported.

The RAM files are much like special relative files in a RAM disk. You have direct access to any place within the file at lightning speeds, but not with the normal **READ FILE** and **WRITE FILE** commands. There are a special set of RAM file package commands. This is unfortunate for several reasons. No current COMAL programs can use RAM files in place of disk files to increase speed. No programs which use RAM files on the c128 will run under any other version of COMAL. A better use of the second bank of memory would be to have a true RAM disk, or to use one bank of memory for the program and the other for the variables (as in UniComal IBM PC COMAL).

The editor appears to use the BASIC 7.0 control codes. All the familiar commands such as removing a line's indentation or erasing to end of line remain, but different keys are used. Lines can be inserted at the cursor position, automatically renumbered as needed. You can even scroll a program listing up or down.

Many commands have been added to c128 COMAL that will be familiar to **Super Chip** users, although most names have been changed. There are commands to peek and poke the vdc registers, to choose the 40 or 80 column screen for output, and to change the CPU speed. **GET$** from an rs232 file has been fixed to prevent an infinite wait. A nice new command allows the user to define the ASCII character translations for files opened with the **/a+** attribute.

User defined packages have been changed in c128 COMAL. Program lines that call a package procedure are the same, but the packages themselves **LINK** into a different section of memory. Packages for the c64 in linkable form (ie, the "*pkg.name*" files usually distributed) will **not** work with c128 COMAL. Worse yet, if a package is linked to a program, the program will almost certainly crash when loaded or **RUN**. However, UniComal provides a new **comsymb** file with the correct addresses for the c128. **If** you have source code for a package, **and** it uses **comsymb** as a library file (rather than defining each label separately), you should in theory be able to reassemble the package to run on the c128 without any major editing (if you have Commodore's Assembler, which we hear is no longer being sold).

C128 COMAL also switches the meaning of the comma and semicolon in print statements. The semicolon positions the text to the next tab as set by the **ZONE** statement. It no longer prints a space. Also, **"" IN "abc"** returns 0 in the c128 version, rather than 4, as in c64 COMAL. These changes may seem minor, but they are just enough to cause programs to work incorrectly.

We will be selling the C128 COMAL cartridge, but we don't have the staff to fully support it at this time. However, if you are interested in programming, the C128 COMAL cartridge offers a fantastic array of features and may be a good choice.

Warning: the c128 COMAL cartridge uses three 32K eproms and seems to be a heavy power drain on the c128. Some of the demo programs sent to us with the cartridge crashed; the computer had to be reset. We don't know if the problem is in the cartridge, the programs, or the computer (most likely it is due to the added power requirement of the cartridge and a border line power supply). ■

CP/M COMAL Preview

CP/M COMAL runs under CP/M 2.2 or 3.0. We have a near final version up and running on a Commodore C128, Epson, and Kaypro. These notes are based on our preliminary version.

The CP/M COMAL manual assumes you have a CP/M computer and that you know how to boot up CP/M. The first thing you need to do is to run the install program to set up COMAL to work with your particular CP/M system. We expect to have the necessary install data included on the CP/M COMAL disk for the more popular CP/M machines, including the Kaypro, Epson and Commodore 128. If you have one of these machines, just choose the file for your machine. If there is no file for your machine, you will need to use your owner's manual to look up the codes to perform tasks such as clearing the screen or turning on underline mode.

Once COMAL is installed for your machine, no more knowledge of CP/M is required, although it may be helpful for advanced applications. Just type **COMAL** to enter the COMAL system.

The COMAL Editor

CP/M COMAL provides a full screen editor, similar to C64 COMAL. You will have the ability to move the cursor up, down, right, or left. You can move the cursor to the top left corner of the screen and optionally clear the screen. You can insert a line between two other lines, delete the line the cursor on, or erase a line from the cursor position to the end of the line. If your system provides these tasks, use the keys you are familiar with. If not, the install program lets you choose the keys to do these things.

Program Entry

Programs should be typed into COMAL using the **AUTO** command. Type **AUTO** and COMAL automatically supplies line numbers for you. Then type in the program line and hit «*return*». At this time COMAL reconstructs the line you just typed. It capitalizes keywords and indents structures for you. If there was a syntax error in the line, COMAL tells you what it was and you must fix it before you proceed. Then COMAL supplies the next line number for the following line. To exit the **AUTO** mode, hit the «*esc*» key.

It is possible to cursor away from the current line to a previous line. For instance, if you discover you misspelled a word in a **PRINT** statement, you can cursor to that line, correct the mistake, and hit «*return*» to enter the correction. You don't need to type the entire line over again or use a special EDIT command. Then you can go back to the current line.

COMAL - The Language

CP/M COMAL includes the commands and structures of standard COMAL (it is amazingly similar to UniComal's C64 COMAL), plus a few enhancements of its own. Its data variable types include floating point numbers, integers, and strings of a user defined length. It is possible to have arrays of each of these types in one or more dimensions with user defined limits on the dimensions. COMAL allows you to print to any output device including random access disk files. Protected INPUT fields are also included. The program environment is interactive, easily accepting user input by a variety of methods.

CP/M COMAL includes **CASE**, **IF**, and **TRAP** structures, plus the four basic loop structures: the **WHILE** loop, **REPEAT** loop, **FOR** loop, and **LOOP** loop. The **IF** and loop structures may be either single or multi-line structures, allowing several statements inside. CP/M COMAL also adds two new features to COMAL's standard loops. An extension of the **LOOP** structure allows it to execute a specific number of times. For example: **LOOP 4 TIMES**

CP/M COMAL has full support of procedures and functions. They may be open or **CLOSED**, allowing both local and global variables. Variables may be passed by reference or by value (a procedure can change the global value of a reference parameter, but not a value parameter). Procedures and functions may be recursive (they can call themselves). Nesting of procedures and functions is allowed. They also may be **EXTERNAL**, called from disk as a program is RUN.

New Functions

CP/M COMAL adds several new built in functions. You may **ROUND** a number or **TRUNCate** it. Bit functions are supported for **and, or** and **exclusive or**. You can **DPEEK** and **DPOKE** two consecutive memory locations. The most useful new functions may be for trigonometry. There are commands for the **arcsin** and **arccos** of an angle. You may also choose to work directly with degrees instead or the standard radians. String functions have been added to convert a number to hexidecimal or binary notation. Strings may also be converted to all uppercase or lowercase characters.

Program Debugging

Two important commands included in CP/M COMAL are **TRACE** and **SYMBOLS**. **TRACE** allows you to single step through a program. It lists each line as it executes to a user defined device and dumps the values of all intermediate calculations. The **SYMBOLS** command prints the names of all active variables and what their current type is.

Packages and Machine Code

CP/M COMAL allows the programmer access to the machine at several levels. There are commands to store or retrieve a value directly from each of the CP/M registers. You may **CALL** a machine language routine starting at any address. You may include in-line machine code to execute short, relocatable routines. Advanced applications should use packages, (similar to UniComal's packages). They support full parameter passing with procedures and functions called by name.

Program Storage

There are two ways to store a CP/M COMAL program. There are **SAVE** files which efficiently store a program in its tokenized format. **LIST** files are stored in true ASCII (not Commodore ASCII) format and may edited by word processors. This is the file format necessary to transfer COMAL programs from one machine to another. We have successfully used the **Big Blue Reader CP/M** to transfer COMAL programs in LISTed ASCII format between IBM, CP/M and Commodore COMALs.

Run-Time System

COMAL users have been waiting years for a run-time system which allows COMAL programs to run without the COMAL interpreter. A utility is available which takes a CP/M COMAL program and produces a **.COM** file which can be run directly from CP/M mode. You can give this program to anyone, even someone who doesn't own COMAL. Note: the **.COM** file is not truly compiled, the program is still interpreted and runs at the same speed as it would under COMAL.

Availability

The final release of CP/M COMAL should be available by June 1987, and we expect to be the USA and Canada distributor. We anticipate the price to be well under $100 (US funds); perhaps $50. The next issue of *COMAL Today* will have further information about CP/M COMAL and its status. We also will post a notice on QLink as soon as we have the final version ready to ship. ■

Disk Sleeve Directories

Super Chip Programs Disk

50 Files — 228 Blocks Free:

hi	chip.disk'editor	vdc'editor.pop	proc.free'disk	~ send sase to ~
menu	chip.draw80col	~~~~~~~~~~~~~~~~	proc.pageline	~~~~~~~~~~~~~~~~
~~	chip.modem	~ function ~	proc.view'bam	~ comal users ~
~ superchip ~	chip.primefactor	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	~ group, usa ~
~ programs ~	chip.quicktest	func.free'disk	~ data files ~	~ 6041 monona ~
~~	chip.sorting	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	~ madison, wi ~
1541'only'editor	chip.testchip	~ procedures ~	dat.sample'text	~ 53716 ~
chip.1000primes	chip.vdc'editor	~~~~~~~~~~~~~~~~	txt.disk editor	~~~~~~~~~~~~~~~~
chip.airplane	smart'reader	proc.center40	~~~~~~~~~~~~~~~~	~(608)222-4432 ~
chip.demochip	sorting'demo	proc.center80	~for more info ~	~~~~~~~~~~~~~~~~

Super Chip Programs Disk

Richard Bain
Marcel Bokhorst
Bert Denaci
Dick Klingens
Len Lindsay
Susan Long
Bob McCauley
David Stidolph

Super Chip on Disk

90 Files — 61 Blocks Free:

hi	~ load ~	~ (rommed) ~	txt.keyboard	~ any of the ~
~~	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	txt.math	~ files on it ~
~comal programs~	~rabbit package~	pkg.superchip-rm	txt.rabbit	~in user group ~
~~	~ (rommed) ~	~~~~~~~~~~~~~~~~	txt.strings	~ libraries or ~
chip.1000primes	~~~~~~~~~~~~~~~~	~ text files ~	txt.system2	~ bbs systems ~
chip.airplane	~ this package ~	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~
chip.demochip	~will not work ~	txt.c128 support	~copyright 1986~	~~~~~~~~~~~~~~~~
chip.primefactor	~with the grey ~	txt.disk notes	~ comal users ~	~for more info ~
chip.sorting	~(eprom) cart. ~	txt.notes-ct#14	~group, u.s.a.,~	~ send sase to ~
chip.testchip	~~~~~~~~~~~~~~~~	txt.notes-ct#15	~ limited ~	~~~~~~~~~~~~~~~~
link'packages	pkg.rabbit	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	~ comal users ~
menu	~~~~~~~~~~~~~~~~	~ quick ~	~ you may make ~	~ group, usa ~
vdc'editor.pop	~ superchip ~	~ reference ~	~ copies for ~	~ 6041 monona ~
~~	~ (non-rommed) ~	~ text files ~	~ your own use ~	~ madison, wi ~
~comal packages~	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	~ only ~	~ 53716 ~
~~	pkg.superchip-nr	txt.c128	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~
~ use the link ~	~~~~~~~~~~~~~~~~	txt.colors	~ do not put ~	~(608)222-4432 ~
~ command to ~	~ superchip ~	txt.files	~ this disk or ~	~~~~~~~~~~~~~~~~

Super Chip on Disk

Phil Bacon
Richard Bain
Marcel Bokhorst
Terry Ricketts
Robert Ross
David Stidolph
John Zacharias

Super Chip Source Code

74 Files — 148 Blocks Free:

hi	src.c128-part2	~ program ~	~ text files ~	~group library ~
~~	src.c128-part3	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	~or bbs system ~
~comal programs~	src.c128-part4	~ use prog'ram ~	txt.assembling	~~~~~~~~~~~~~~~~
~~	src.c128-keypad	~ to turn into ~	txt.c128graphics	~for more info ~
prog'ram	src.system2	~binary package~	~~~~~~~~~~~~~~~~	~ send sase to ~
change'autostart	src.math	~~~~~~~~~~~~~~~~	~this disk and ~	~~~~~~~~~~~~~~~~
~~	src.strings	~ use ~	~ the source ~	~ comal users ~
~ superchip ~	src.keyboard	~ change'auto ~	~code on it is ~	~ group, usa ~
~ source code ~	src.colors	~to change load~	~ copyrighted ~	~ 6041 monona ~
~~	src.files	~address $5000 ~	~ 1986 ~	~ madison, wi ~
src.chip-nolist	src.rabbit-1	~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~	~ 53716 ~
src.chip-list	src.rabbit-2	autostart	~you cannot put~	~~~~~~~~~~~~~~~~
sym.comal	src.rabbit-3	lst.autostart	~ this disk or ~	~(608)222-4432 ~
src.header	~~~~~~~~~~~~~~~~	bin.autostart	~ the files on ~	~~~~~~~~~~~~~~~~
src.c128-part1	~ autostart ~	~~~~~~~~~~~~~~~~	~it in any user~	

Super Chip Source Code

Phil Bacon
Richard Bain
Marcel Bokhorst
Terry Ricketts
Robert Ross
David Stidolph
John Zacharias

CT#16 - 0.14

Phyrne Bacon
Richard Bain
Jack Baldrige
Lewis Brown
Gerald Hobart
Sol Katz
Len Lindsay
Greg Mavko
Bill Nissley
Charl Phillips
Sid Seiflein
David Stidolph
Herb Wollman

Today Disk #16 - Front

57 Files | 4 Blocks Free:

boot c64 comal	----------	read'bytes	----------	----------
c64 comal 0.14	2'drive'copier	read'directory	- data file -	- comal users -
comalerrors	basic2comal-p1	see'index	----------	- group, usa -
ml.fastload-msd	basic2comal-p2	shadow'letter1	ran.startrek	- 6041 monona -
ml.sizzle	cubic'spline	shadow'letter2	read and run.txt	- madison, wi -
hi	demo/magicsquare	smart'reader	----------	- 53716 -
menu	dirprint'tiny	smooth'curve	- function -	----------
comal article	edit'box	sprite'fun	----------	-(608)222-4432 -
info.txt	file'numbers	star'trek'db	func.graphics'on	----------
names.dat	graph'fp'error	subtract	----------	
----------	mad's-alfred'e	the'15'puzzle	-for more info -	
-comal programs-	magic'square	uniform'spline	- send sase to -	

CT#16 - 2.0

Phyrne Bacon
Richard Bain
Jack Baldrige
Thomas Bishop
Marcel Bokhorst
Reed Brown
Captian COMAL
Joel Ellis Rea
Gerald Hobart
Bill Inhelder
Russell Jensen
Dick Klingens
Gerald K. Hobart
Len Lindsay
Susan Long
Robert Ross
Robert Shingledecker

Today Disk #16 - Back

79 Files | 1 Blocks Free:

hi	editor-cai	- packages -	- procedures -	sample.f-for
----------	fast'fourier	----------	----------	sample.topr
-comal programs-	graph'fp'error	pkg.matrix	proc.command	sample.data
----------	magic'square	pkg.meta	proc.information	----------
cbase	make'dummy'files	pkg.mlw'sorts	proc.merge'text	-for more info -
chip.rabbit'test	matrix'calc	pkg.no'pps	proc.nofont	- send sase to -
customize'list	nim	pkg.screenhelp	proc.print'using	----------
demo/infomaker	print'using	pkg.text	----------	- comal users -
demo/magicsquare	risk'dice	----------	- functions -	- group, usa -
demo/matrix	see'index	- this file -	----------	- 6041 monona -
demo/no'pps	shorten'bat'file	-requires meta -	func.cart'color$	- madison, wi -
demo/screenhelp	smart'reader	- package be -	func.roll'dice	- 53716 -
demo/sorting	smooth'curve	-linked to use -	----------	----------
directory'probe	star'trek'db	----------	- data files -	-(608)222-4432 -
dirprint'tiny	subtract	lst.tron'w/meta	----------	----------
disk'editor	----------	----------	bat.2drive'keys	

Math / Science Disk

Richard Bain
Doug Colpitts
David Eagle
Kurt Heinrich
William Inhelder
Richard J. Shaggot
Jim Kaminski
Todd P. Shaggot
Lowell Zabel

Math / Science Disk

40 Files | 3 Blocks Free:

hi	galilean	----------	pkg.matrix	----------
----------	its'elementary	- comal 2.0 -	----------	- comal users -
-comal programs-	noon sight	- package -	- data file -	- group, usa -
----------	primary'math	----------	----------	- 6041 monona -
celestial'nav	rocket1	- do not load -	dat.stars'mag	- madison, wi -
curvefit	sky'view	----------	----------	- 53716 -
demo/matrix	star'mapper	- use link -	-for more info -	-(608)222-4432 -
fast'fourier	syncsat	----------	- send sase to -	----------

2.0 Data Base Disk

Bob Hoerter
Russ Jensen
Len Lindsay
Charl Phillips
Robert Shingledecker

2.0 Data Base Disk

88 Files | 1 Blocks Free:

hi	db'data	- data base -	- text files -	- require the -
menu	db'help.def	----------	----------	- comal 2.0 -
----------	db'help.lab	- len lindsay -	txt.about comal	- cartridge to -
- data base -	db'help.rpt	- and -	txt.cbase	- run -
- manager -	db'name	-charl phillips-	txt.db'tutorial	----------
----------	----------	----------	txt.star trek	-for more info -
- robert -	- video filer -	star'trek'db	txt.video filer	- send sase to -
-shingledecker -	- system -	ran.startrek	----------	----------
----------	----------	----------	-these programs-	- comal users -
db'boot	- bob hoerter -	- cbase -	-must be moved -	- group, usa -
db'define	----------	----------	- onto another -	- 6041 monona -
db'help	videofilersystem	- russ jensen -	- disk before -	- madison, wi -
db'labels	ext.correctfiles	----------	- being used -	- 53716 -
db'maintenance	ext.enter'record	cbase	----------	----------
db'menu	ext.split'file	sample.f-for	- all programs -	-(608)222-4432 -
db'report	ext.sub-file	sample.topr	- on this disk -	----------
db'sort	----------	sample.data	-are written in-	
db'squash	- star trek -	----------	-comal 2.0 and -	

Mytech IBM PC COMAL Preview

Mytech IBM PC COMAL 2.0 runs under MS-DOS or PC-DOS 2.0 or greater. These are our notes after testing a nearly final release on our Zenith IBM PC compatible.

Line Editor

The line editor is similar to the editor provided by MS-DOS. It is useful for quick program editing, such as inserting a line here and there. If you make a syntax error in the edit mode, COMAL will tell you to what it is, so you can fix it before moving on. If you aren't sure what correction is needed, distinct help screens are available for many keywords and program structures. The line editor includes an **AUTO** mode to provide line numbers if desired. However, this is not the recommended method for program entry. It won't let you cursor up or down to a line already displayed on the screen. You must use the full screen editor for that type of editing.

Full Screen Editor

The full screen editor is more like a specialized word processor for programs, a programmer's dream. It is the best of any COMAL we have seen, and includes help screens. You have full cursor movement in any direction. Control keys allow you to move to various positions within the text. You can move to the start or end of line, or to the start of the program. You can move one word forward or backwards within a line. For longer programs, you can move forward or backward a screen at a time without having to list a new line range. There are options to delete the line the cursor is on or to delete the current line from the cursor position on. You can insert a blank line between two other lines without ever needing to renumber. Line numbers are not shown on the screen.

To enter a new program line, or alter an existing one, you merely type the line or changes. There is no need to hit «*return*». Just move your cursor off the line you were working on, and it is automatically entered for you. Once lines have been entered, they can be moved. It is simple to highlight a procedure, pick it up and move it to a more logical position within the program. You also have the option to duplicate common sections of your program code into other areas.

Each time a line is entered, it is properly relisted. Keywords are capitalized, = is converted to :=, **THEN** and **DO** are added as needed, and so on. When you move down to a blank line, COMAL automatically indents that line for you, tabbing the cursor to the proper starting location. A control key can be used to scan the program and insert the variable names for **ENDPROC**, **ENDFUNC**, and **ENDFOR**.

Scanner

Before you run a program, you may scan it for structure errors. It knows that a **REPEAT** loop does not end with **ENDWHILE**, and much more. If you call a procedure or function which is not defined in the program, the scanner will tell you. It even knows if a parameter list is correct, and will warn you when it is not.

Full 2.0 Features

Mytech COMAL is very similar to the other implementations of COMAL 2.0. It includes **CASE**, **IF**, and **TRAP** structures as well as **REPEAT**, **WHILE**, **LOOP**, and **FOR** loops. Procedures and functions may have reference and value parameters plus local or global variables. They may call themselves recursively. They also may be **EXTERNAL** (disk loaded).

Scope

Mytech COMAL has one major difference from the other implementations of COMAL. Most COMALs have dynamic scope rules (*see COMAL Today #14, page 60*). Mytech COMAL has static

more»

scope rules. This is an advanced programming topic that is unimportant to most COMAL programs (and programmers). However, for the few programs which take advantage of scope rules, there are subtle differences. Note that dynamic and static scope rules are different, but that neither is "better". (*Pascal and most of the other Algol based languages have static scope rules. LISP is an example of a language using dynamic scope rules.*)

Debugging

Once the program is written, it is time to debug it. Mytech COMAL has a variety of options to trace program flow. You can single step through the entire program or single step through a specified range of lines. One single step mode lists each line as it executes, stops the program, and waits for you to press «*return*» before continuing. The other single step mode lists each line as it executes without stopping the program. An advanced option for either single step mode also prints out the values of all variables and temporary calculations used during each line. This makes it fast and easy to track down any logical errors in a program. After debugging the program, there are no trace commands to delete because all line ranges are set from the immediate mode, not as program statements.

Trapping the Stop Key

The «*stop*» key («*control*» «*break*») can be enabled and disabled anywhere within a program. A third option has been added to make the «*stop*» key produced a trappable error. This can cause a jump to a routine to **CLOSE** files and return output to the screen, or perhaps a jump to the program's main menu. This technique removes the need to continually check the **ESC** function, and will work during INPUT requests.

Array Handling

Mytech COMAL has added several new array commands. Entire arrays can be read, written, or printed at once, without the need for a **FOR** loop. **Functions may RETURN arrays**, and one array can be assigned to another. There are even commands to determine what the minimum and maximum elements are for each dimension in any array.

File Handling

In addition to the standard COMAL file commands, Mytech has added a few new file commands as well. A notable one is a function to determine an unused file number. Generic functions such as a *file'exists* function no longer need to worry what file numbers the main program happens to be using. Other functions tell how much storage space is required for the various data types.

Graphics and Other Packages

The user may write COMAL packages using the Whitesmiths C compiler (other C compilers soon, we hope). The graphics package was not ready for our preview. The 8087 Math Co-processor chip is supported, but not required.

Availability

The final version should be ready to ship by June 1987. We expect to be the USA and Canada distributor. The price should be under $100, complete with manual.

A special preliminary release DEMO disk is available now for $5. As soon as the final COMAL 2.0 is released, we also should have a new updated DEMO system. This DEMO system is the full COMAL, without a SAVE command.

Mytech informs us that their COMAL has been officially approved by IBM in Sweden. ■

C64 COMAL 0.14 & 2.0

There are two different COMAL implementations running on the C64 (0.14 and 2.0). A popular questions is *"What is the difference between them?"*

The most obvious difference is that COMAL 0.14 is disk loaded with about 12K free programming space. COMAL 2.0 is a cartridge with 30K free for programs. Beyond this distinction, they are two dialects of the same language; both written by UniComal.

Both versions contain a variety of program structures including an **IF** statement to conditionally execute sections of code, and a **CASE** statement to choose a section of code based on a multiple choice test. There are three types of loop structures. The **FOR** loop executes lines a specified number of times. The **WHILE** loop and **REPEAT** loop execute an indefinite number of times. The difference is when the exit condition is tested. All of these structures can be multi-line, allowing several statements inside the structure. Some have one-line counterparts to save space when only one statement is needed inside the loop.

There are several built in procedures and functions. These include support for turtle graphics, sprites, mathematics and more. The programmer can define other named procedures and functions to allow large sections of code to be called with one descriptive statement. Variables may be passed as parameters to the procedures. Internal procedure variables may be local to procedure or global to the entire program. Long variable names make this even more convenient.

Beyond the structure, there are also friendly input/ output routines. These include commands for user input with screen prompts, and support for sequential and random access files. The standard **PRINT** statement may be redirected to the printer if desired.

COMAL programs are easy to type in. An automatic syntax checker ensures that a line is correct before it is accepted. Otherwise, it gives an informative error message. COMAL also checks if a program is structurally correct, before it lets you **RUN** it.

If both versions have all that, why should anyone spend buy the COMAL 2.0 cartridge? Many don't need to, but you must decide for yourself. The cartridge is easier to use. It adds more programming power and memory and is needed to run our best programs. Yet 0.14 is more than enough for many casual programmers.

COMAL 2.0 has everything that COMAL 0.14 has and more. It is more compatible with the versions of COMAL running on other computers. It fully implements the COMAL Kernal, and then adds many enhancements: **PAGE** to clear the textscreen; **PRINT AT** to place text anywhere on the screen; protected **INPUT** fields; **GET$** gets characters directly from a file without worrying about the format; and **STR$** and **VAL** to convert numbers from internal floating point format to a character string and back. It also adds the **LOOP** structure, with the **EXIT** condition in the middle.

Other luxury features include definable function keys, useful control key commands for editing, text and graphics screen dumps to the printer or a disk file, better error messages, built in modem communication capability, and automatic true ASCII conversion. In addition, the cartridge has **major** extensions:

FIND and **CHANGE** can be used from the editor to do what their names imply. They allow you to type in short variable names and then change them to longer more descriptive ones.

TRACE. If your program isn't working right, you can stop it and type **TRACE**. COMAL will tell you how it got to the point where it stopped.

more»

TRAP / HANDLER can be used to prevent programs from crashing when an error condition occurs. If a program needs to access a disk file, the program can give the user a second chance to put the correct disk in the drive, rather than ending with a disk read error.

Batch files. Now your C64 can execute commands from a disk file as if you were typing them! They work from immediate mode or program mode.

External procedures. Now you can store your procedures and functions on disk. Other running programs can use them as needed. Sort of an automatic overlay system. With just one disk drive, you can have an extra 170K of procedures which are only stored in memory when they are running.

Packages. A package contains additional commands (written in machine code) that can be *attached* to COMAL. The cartridge comes with 11 built in packages for graphics, sprites, sound, a light pen, and more. You can create your own set of disk loaded package commands. **LINK** them to any program! Dozens of packages are already available, ready to **LINK** and use.

Empty socket. Packages can be *burned* into an EPROM and inserted into the empty socket inside the cartridge. This allows you to have your own built in added commands. An example of this is Super Chip, which adds over 100 new commands to the cartridge.

Another concern is whether a program written for one version will run in the other. Most programs require only a few minor changes. These changes are described in detail on page 42 of *COMAL Today #12.*

[User Groups: feel free to copy or reprint this article in your own newsletter. It may answer questions that many of your members have.] ■

UniComal IBM COMAL

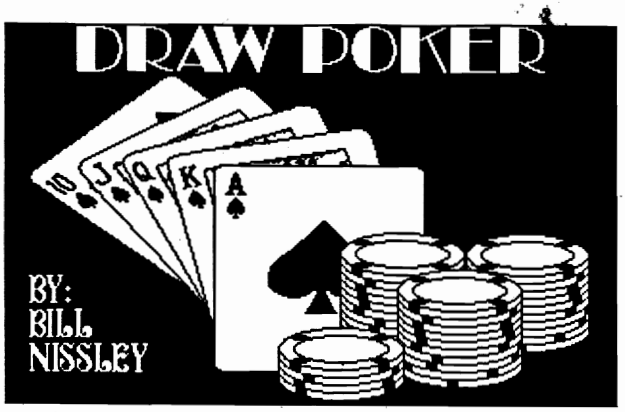
UniComal's COMAL 2.0 for the IBM PC and compatibles is very similar to their C64 COMAL 2.0 cartridge, right down to the full screen editor. If you are a cartridge user, you will feel right at home with UniComal's IBM PC COMAL.

The original 2.0 release is distributed by IBM Denmark with a Danish manual. But recently, UniComal released an updated 2.1 version with an English manual and sell it themselves. Just last week we heard that IBM Denmark is still distributing the original 2.0 release. We can special order either version for you (we assume you want the new 2.1 release unless you specify "original" release).

The new release adds the TRACE command, which we missed in the original. It also includes a command to list all the package keywords. This is very useful. It also is faster, and includes added EGA support.

However, the new 2.1 release has some minor incompatibilities. The meaning of the comma and semi-colon in a print statement is switched. The comma now always gives one space. Also, the null string with the IN comparison now has an opposite result (ie, UNTIL key$ IN "YyNn"). These two changes can lead to subtle errors in programs (or down right failures). Therefore we are continuing to use the original 2.0 release to run our Order Processing System.

Both 2.0 and 2.1 are very fast and support the 8087 math chip. Special editing commands are accomplished with control keys, and the function keys are definable with an optional display on the bottom screen line. We especially like the built in text screen windowing and extra long program lines (up to 240 characters per line). Free memory is divided into two 64K blocks, one for the program, the other for variables. This is the maximum, even if your computer has 640K available. ■



After a recent trip to Las Vegas I came home and wrote a program based on the latest craze of one arm bandits. *Draw'poker* is on *Today Disk #17*. The biggest differences are the video screens, not needing to pull the arm, nor to handle those dirty coins. These new machines play off credits you build up. Of course you have to put some money in to begin with, but then it's a no muss no fuss operation after you build up some credits. You collect your winnings with the push of a button.

The major differences in my program are you start with ten dollars in the machine and you don't get to cash in your winnings when you call it quits (*I never seem to be able to call it quits until I'm broke anyway*).

The controls are simple, at the beginning and after each hand, you place your bet of 1 to 5 coins by typing the number and pressing «*return*». The number selected and the payoff for the different hands are highlighted at the top of the screen.

200 to 1 - Royal flush
50 to 1 - Straight Flush
25 to 1 - Four of a kind
10 to 1 - Full House
8 to 1 - Flush
5 to 1 - Straight
3 to 1 - Three of a kind
2 to 1 - Two pair
1 to 1 - A Pair of Jacks or better

After placing the bet you are dealt five cards. Use the keys 1-5 to select which cards you wish to hold. They will toggle back and forth so be careful all the cards are set the way you want before you press «*return*».

The new cards are then dealt and the hand is checked for a winner. The payoff is made to your credits, the high score is updated (if needed) and the amount of coins you won (if any) is shown on the right. While the payoff is being made, the hand you got is highlighted at the top (in case you don't know why you won).

Place a bet and a new hand is dealt. This goes on until you go broke. The title screen comes back on and starts the game over (unlike Vegas you get $10 free credits every time you bust).

This program uses the text screen for game play with a special font for the cards and the 25 cents symbol at the bottom right. The title screen is a hires screen made using *Print Shop* and *Print Shop Companion* for the lettering, *ComputerEyes* digitizer for the cards, *Doodle* for putting it together and creating the chips and a program I wrote to bring it all together so COMAL can use it.

The trickiest part for me to do in this program was to have the computer judge the final hand. These routines were done with the help of the *yahtzee* program from *Today Disk #9*. The procedures were a great help, but had to be modified to account for larger numbers, face cards and the ace which can go on either end of a straight. Also new procedures were made to check the suit of the cards (not needed with dice). To check these out list the procedure **judge'hand** to find the procedures involved.

The other part that I thought was going to give me a hard time was the shuffling of the deck after each hand (no card counting with this game). I soon realized that the computer didn't need to shuffle as mere mortals do because it can randomly deal from anywhere in the deck, Some mortals can do that but they better not get caught. The computer does this without trying to deal you a bad hand. ■

Underline Cursor 

by Dick Klingens, Dutch COMAL Users Group

As you perhaps know, the cursor on our C64 screen isn't a real cursor. It is created by repeatedly showing the character at the cursor position in its normal and reversed shape.

We illustrate the routine in COMAL:

```
PAGE
screen:=peek(648)*256
row:=12; col:=20
offset:=(row-1)*40+(col-1)
address:=screen+offset
code:=1 // screen code for A
LOOP
  POKE address,code
  delay
  POKE address,code+128
  delay
ENDLOOP // forever
//
PROC delay CLOSED
  FOR w:=1 TO 500 DO NULL
ENDPROC delay
```

This puts a blinking A in the middle of the screen. The normal/reversed shapes are stored somewhere in memory and can simply be changed using the **Font** package. We shall use that package to create an underscore cursor.

Each character is stored using 8x8 bits. For example the character A:

normal	reverse	underline
00011000	11100111	00011000
00111100	11000011	00111100
01100110	10011001	01100110
01111110	10000001	01111110
01100110	10011001	01100110
01100110	10011001	01100110
01100110	10011001	01100110
00000000	11111111	11111111

The 8th line in the normal shape is not in use. We change this line into **11111111** and use this new shape in place of the reversed one. In this way we have the impression of an underscore cursor. The following procedure changes the reversed characters in the fonts 0 to 1 into underlined characters. Issue these two commands from the immediate mode:

```
USE font
linkfont
USE font
keepfont
```

This puts the ROM character set into RAM that can be changed by the user.

```
PROC set CLOSED
  USE font
  DIM kar$ OF 8
  FOR ch#:=0 TO 127 DO
    FOR fnt#:=0 TO 1 DO
      getcharacter(fnt#,ch#,kar$)
      kar$(8):=chr$(255) // set underline
      putcharacter(fnt#,ch#+128,kar$)
    ENDFOR fnt#
  ENDFOR ch#
ENDPROC set
```

To change the underscore cursor back into the normal one we must use a trick, because a kept font cannot be discarded. So type:

```
nofont
discard
```

using the following procedure:

```
PROC nofont CLOSED
  POKE $c2bb, PEEK($c2bb) BITAND 127
ENDPROC nofont
```

Set'cursor on *Today Disk #17* demonstrates the above commands. ■

more»

Help Screen Editor 

by Dick Klingens, Dutch COMAL Users Group

In *COMAL Today #15* there was an article about the use of the **Text** package. It describes how to save *help* screens with your programs. However, creating such help screens is something else. I'll describe two possible ways to do so.

First type the following program:

```
PROC to'file(n) CLOSED
 USE system
 DIM screen$ OF 1505
 getscreen(screen$)
 OPEN FILE 9,"scrn.help"+STR$(n), WRITE
 WRITE FILE 9: screen$
 CLOSE
ENDPROC to'file
```

SCAN this procedure before you continue.

Now clear the textscreen and type in edit mode the screen with the help text. This work has to be done very carefully, because:

1) you can not use the return key.
2) typing a character in the 40th column of the screen sometimes inserts a blank line.
3) typing on the 25th line of the screen can scroll the screen.

In all these cases you have to do the whole job again. When finished (at last), type in background color on a free line the command

to'file(1)

The screen is saved now. Re-editing such a screen is possible but you can damage your work easily.

Of course, we need a **help screen editor**: The second way. I developed such a program; *Makescreen* is on *Today Disk #17*. Creating *help* screens with this program is much easier. There are some very useful features in the program.

The editor starts with drawing a frame around the screen. You can change the textcolor with the usual «*ctrl*» or «*C=*» plus number key combinations. Border and background colors can be set with «*f3*» and «*f4*». Press «*ctrl-f*» to redraw the frame in the current textcolor. «*Ctrl-n*» erases the frame.

Because the screen has a logical line length of 80 characters, typing a character sometimes inserts a blank line (I mentioned this fact above). That is why there is a special way to place a character in the 40th column (in the frame). For logical reasons I used the same process for a character in the first column. Place the wanted character in the 39th or 2nd column and press «*f6*» or «*f54*» and the character moves into its correct position. Note: the frame and no frame commands overwrite text in column 39.

With the cursor keys one can set the edit direction. For instance, press «*left*» once and future typing will be displayed from right to left. It is also possible to type from top to bottom or bottom to top. It may be necessary to type the «*right*» cursor key to restore typing to the normal left to right direction. Press «*return*» to continue editing on the following line.

«*Rvs on*» is used to turn on the reverse (highlight) mode. Any characters typed after this command will be highlighted. If you cursor over a character already typed, it will be highlighted too. Press «*rvs off*» to turn off the reverse mode.

«*Ctrl-d*» and «*ctrl-l*» are for saving (Dump) and Loading a screen to or from disk.

With «*f1*» one can temporarily store a screen; «*f2*» restores that screen.

more»

Help Screen Editor - continued

Of course the program has its own help screen saved with it. It is invoked by pressing «*f7*».

Quitting the program is possible in two ways:

«*f8*»: quit, but save the screen into the file *scrn.screenmade*.
«*stop*»: quit, but do not save the screen

Scrn.makehelp contains the help screen of the program. You can change this screen as you like it.

In the program there is a special procedure to replace the original help screen with this one. Type in edit mode:

to'text
DELETE "makescreen"
SAVE "makescreen"

I am sure that *makescreen* makes it easier for you to design your help screens.

A final remark.

Makescreen can also serve to create screens used at program startup, especially when these screens are difficult to build up in the program itself. ■

```
MAKESCREEN
[f1/2]   - Get/Set actual screen
[f3/4]   - Change border/backgr color
[f5]     - Push char in col 2 left
[f6]     - Push char in col 39 right
[f7]     - HELP (toggle)
[f8]     - Dump scrn in file & Quit

[home]   - Cursor home
[stop]   - Quit program (no Save)
[crsr]   - Toggle edit direction
[ret]    - Edit on next line
[clr]    - Get screen & Clear screen
[color]  - Change textcolor
[rvson]  - Reversed text ON
[rvoff]  - Reversed text OFF

[ctrl+f] - Redraw frame in text color
[ctrl+l] - Load screen from file
[ctrl+d] - Dump screen to file

by D.Klingens-Dutch COMAL Users Group
```

Batch to Package

by Jack Baldrige

Not long ago, I was loading a batch file and wishing that I could keep it in memory, where it could be kept for repeated use without disturbing any programs. The 'memory' part rang a bell, so I dug into my back issues of *COMAL Today* and reread Jesse Knight's article *Batch Files From Memory* in *COMAL Today #7*, page 32. After I read his explanation of how the function keys are handled by COMAL 2.0, the way to do what I wanted was pretty straightforward.

Batch'to'package makes a linkable package from a batch file. You input the name of the batch file and the name you want for the package file, as well as the name of the package itself. The package may start almost anywhere from $6000 to $bfff. You may enter the address in decimal or hex. The package may be activated when linked or only after the command **USE packagename**.

While it is convenient to keep a batch file in memory, I hadn't considered another feature of such a batch-package, speed. It is much faster to read the batch file from memory than to read it from disk.

There are a couple of things to watch for when using this program. In order to save memory, if two or more consecutive spaces are found in the file, all but one are eliminated. Therefore, your batch files should use some character other than the space as a separator. If you need consecutive spaces, you can use **SPC$**.

Since the package uses COMAL function key routines, a batch-package can't be used to redefine more than one function key. After it defines one function key, the package stops dead! Some might desire to have a package with more than one procedure, but I prefer to leave that job for the reader. *Batch'to'package* is on *Today Disk #17*. ■

Screen Editor Revisited

by Gerald Hobart 

Screenhelp is a combination of machine language code and COMAL 0.14 procedures which together implement eight commands for controlling output to the textscreen. Among other uses, these commands greatly facilitate the addition of windows to your programs. The program is on *Today Disk #17* along with a file of supporting machine language routines, *screen'help.obj*. The commands are described below. *COMAL Today #16* had the COMAL 2.0 version of screenhelp (*pkg.screenhelp*).

Getscreen and **setscreen** work like their counterparts in COMAL 2.0 **System** package. That is, **getscreen** stores the current textscreen and all its attributes (border, background and text colors as well as cursor position). Subsequently, **setscreen** will restore the textscreen. The one difference between this and the COMAL 2.0 version is that these commands store the screen to RAM at 50741-52245 and not to a string variable. Thus only one screen may be stored at a time. (Subsequent uses of **getscreen** store the current screen, but wipe out any previously stored screen.)

Textwindow(<row1>,<col1>,<row2>,<col2>) defines the textscreen window which will apply to subsequent *screenhelp* commands. **<Row1>,<col1>** specifies the upper-left hand corner of the window and **<row2>,<col2>** specifies the lower-right hand corner of the window. Rows are numbered 1-25 and columns are numbered 1-40. Until a **textwindow** command has been given, the default window is the entire screen (1,1,25,40).

Windowframe(<color>) draws a frame around the inside of the currently defined window in the color specified by **<color>**. If one uses this command it is normally desirable to re-define the window slightly smaller afterwards, otherwise the frame itself will be affected by subsequent *screenhelp* commands.

Change'frame toggles the type of frame produced by **windowframe**. A double frame is the default type. **Change'frame** will change it to a thin, single frame. This does not affect frames already drawn, only subsequent frames. If you want to design your own windowframes, you can poke the desired screen codes to the following locations:

50183 - Upper Left Corner
50184 - Upper Right Corner
50185 - Lower Left Corner
50186 - Lower Right Corner
50187 - Top/Bottom Lines
50188 - Left/Right Sides

For example try poking either 127 or 160 to all these locations, then issue a **windowframe** command.

Reverse will *invert* the mode of all material within the current window. That is, all material printed in the normal mode will be switched to the reverse mode, and vice-versa. Thus, using this command a second time will bring the material back to its original state. Since the window can be defined to be a portion of a single row, this command is useful for making a moving-bar type of menu.

Textfill(<char>) fills the current window with the character whose screen display code is specified by **<char>**. (See Appendix B of the Commodore 64 Programmer's Reference Guide for a table of screen display codes.) For example, **Textfill(32)** clears the current window.

Colorfill(<color>) fills the color-ram corresponding to the current window with the color specified. The effect of this is to change the color of all printed characters within the window to the selected color.

No scrolling commands were included. However, this code is completely compatible with either

more»

version of the Window Magic routines by David Warman (*window-rs232.obj* and *window-sprit.obj* on *Today Disk #14*) if scrolling commands are needed. *[David Warman's code is used for the scrolling menu on recent COMAL Today Disks.]*

The machine language code is located from 50176 to 50740 and the **getscreen** command stores the screen from 50741 to 52245. Thus you are restricted to using sprite shapes 0-15 and 49-56.

Prior to using any of these commands your program must load the machine language code into memory with the command:

load'obj("screen'help.obj")

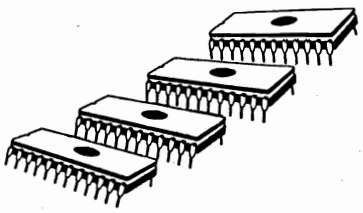
Of course the procedure **load'obj** must also be included in the program, and the file *screen'help.obj* must be on the disk. Then to use any of the commands above, the corresponding procedure must be appended to your program.

Examine and run the program *demo/screenhelp* to see how the commands can be used. I would appreciate hearing about any problems encountered with this code and would enjoy seeing any applications of it. My address is 1203 Thomas Dr., Bellevue, NE 68005. ■

Super Chip On Disk Packages

Super Chip On Disk includes these packages:

Rabbit
Colors
C128
Files
Keyboard
Math
Strings
System2 ■



Super Chip On Disk

Now that so many people have Super Chip, we may be receiving more programs that utilize Super Chip commands. However, any cartridge owner can run those programs if they have Super Chip On Disk! Then before you try to run a program that requires Super Chip, first link in your disk based Super Chip package:

LINK "pkg.superchip-rm"

This is a "*rommed*" package. This means that the package will stay "*hidden*" under the cartridge until the power is turned off. It ignores commands such as **new, discard, load,** and **enter.**

Once you have issued the **LINK** command shown above, you can **LOAD** and **RUN** programs that require Super Chip. This will work for both the beige and black C64 COMAL 2.0 cartridges.

Added Super Chip On Disk Information

All the commands of Super Chip except **autostart** are available from Super Chip On Disk. However, the commands were split into two logical parts: **Rabbit** and everything else. This was done for two reasons. It allows you to link in the **Rabbit** fastloader first. Then Super Chip On Disk code links in much faster.

The other reason was to make Super Chip On Disk more compatible with other packages. **Rabbit** is assembled to start at $8009, where most packages we have released also start, making **Rabbit** incompatible with most packages. Super Chip On Disk starts at $9009, so it can be used with virtually any packages less than 4K which only occupies the $8009-$8fff block. Thus Super Chip On Disk is compatible with nearly all our packages released (but not with **cmon** and **text**). Also note: the C128 package uses the cassette buffer and can interfere with programs that also use that buffer.

Now where are all those Super Chip programs?■

Apple COMAL Notes

In *COMAL Today* #16 I described the Apple COMAL project and how you can get involved. To summarize, I am writing COMAL for the Apple II family of computers. Version 1.0 is for Apples with at least 64k. Version 2.0 is for 128k Apples. I am looking for people who want get involved with its development. Getting involved means testing preliminary versions, responding to questions asked in *Apple COMAL Notes* (a newsletter I am publishing for this project), and giving me your opinions on what you like and don't like. Apple COMAL will run under ProDos only.

Version 1.0 will have most of the features of COMAL 2.0 like nested procedures, local DATA statements, TRAP/HANDLER, IMPORT, protected INPUT fields, PRINT AT, and string functions. I am making 1.0 as advanced as possible so that programs will be truly compatible between version 1.0 and 2.0. Version 1.0 will not have packages or EXTERNAL procedures/functions. This means no graphics support for Version 1.0.

Version 2.0 will have more free memory for programs, EXTERNAL procedures/functions, and packages. It will come with several packages, including turtle graphics.

Apple COMAL Notes

The goal of *Apple COMAL Notes* is to inform the subscriber on how COMAL functions (both internally and how the user perceives it), and provide implementation questions and ideas.

The newsletter is published monthly, and I have published 2 issues so far. The issues are done in 2 column format similar to *COMAL Today*, printed on white bond paper. They are mailed without punching or binding, leaving it to the reader how to store the issues. The newsletter is not written for beginners, it is written for people who already know COMAL, and want to delve further into the language.

The first issue was 10 pages and included an overview of COMAL and its features, along with a technical discussion of how COMAL works.

Issue two was 20 pages and covered the COMAL Kernal as well as a look at memory usage on the Apple and how it affects COMAL.

Please write for further information.

David Stidolph, COMALites United
1670 Simpson #102, Madison, WI 53713
(608) 222-4505 - Evenings only ■

ORDER FORM Subscriber #_____ Name:_____

(more on other side)

April 1987 - Prices subject to change Street:_____

(only subscribers get lower prices)

Prepaid orders only, US Dollars, or charge: City/St/Zip:_____

VISA/MC #:_____exp date:_____Signature:_____

Circle price of items wanted: (first is DocBox pages price, second-subscriber price, third-list price - DocBox is pages only)

BOOKS - Shipping add $3 per book (overseas is extra)

DocBox	Sub price	List price	Item description - disks are Commodore format - DocBox is pages only
	$9.95	$14.95	New-Doc Box, mini 3 ring binder & slip case, cloth bound, quality D-ring style One Doc Box is required to hold books you buy as Doc Box pages!!! (see special below)
$15.95	-.-	$20.95	New-COMAL Cross Reference, Len Lindsay, *estimate over 200 pages* (due July 1987) *General* || Detailed reference book for COMAL 2.0 implementations in USA (CP/M, Mytech, UniComal)
$12.95	$17.95	$19.95	COMAL 2.0 Packages, Jesse Knight, *108 pages with disk* *C64 2.0 Advanced* || How to write a package in Machine Code; includes C64 comsymb & supermon
$12.95	$17.95	$19.95	Packages Library Volume 1, David Stidolph, *76 pages with disk* *C64 2.0* || 17 example packages ready to use, many with source code, plus the Smooth Scroll Editor
$12.95	-.-	$19.95	New-Packages Library Volume 2, *estimate 50 pages with disk* (due July 1987) *C64 2.0* || more example packages ready to use, many with source code
$9.95	$14.95	$15.95	Graph Paper, Garrett Hughes, *52 pages with disk* *C64 2.0* || Function graphing system for C64 COMAL 2.0. The program can't be LISTed.
$15.95	-.-	$20.95	New-COMAL Collage, Frank & Melody Tymon, *about 200 pages with disk* (due July 1987) *C64 2.0* || Graphics and Sprites tutorial with many full sized example programs
$12.95	-.-	$18.95	New- 3 Programs in Detail, Doug Bittenger, *book and disk* (due July 1987) *C64 2.0* || A step by step guide to creating: Blackbook, Home Accountant, BBS
$6.95	-.-	$11.95	New-Today Tutorials, From COMAL Today #1-17, updated as needed (due July 1987) *General* || Includes explanations of structures, procedures, functions, and string handling
$6.95	-.-	$11.95	New-Today Tips & Notes, Collection of programming tips and notes (due July 1987) *C64 0.14 & 2.0* || Taken from COMAL Today issues #1-17, updated as needed
$4.95	$4.95	$6.95	Cartridge Graphics & Sound, Captain COMALs Friends, *64 pages* *C64 2.0* || A reference book to the 11 packages built into the C64'COMAL 2.0 cartridge
$17.95	$22.95	$24.95	Cartridge Tutorial Binder, Frank Bason & Leo Hojsholt, *320 pages with disk* *C64 2.0* || Now available punched for our Doc Box! The tutorial from Commodore Denmark
-.-	$4.95	$6.95	COMAL Today - The Index, Kevin Quiggle, *52 pages with disk,* *General* || 4,848 entry index to COMAL Today issues 1-12, full data disk files & reader program
-.-	$14.95	$16.95	COMAL Yesterday, first four issues of COMAL Today spiral bound *General* || The original issues are out of stock, this is the special reprint
-.-	$17.95	$19.95	Introduction to Computer Programming with COMAL, J William Leary, *272 pages* *C64 2.0* || Beginners text book, spiral bound, now includes separate answer book
-.-	$16.95	$18.95	COMAL Handbook, Len Lindsay, *479 pages* *C64 0.14 & C64 2.0* || Detailed reference book for C64 COMAL 0.14 and 2.0
-.-	$17.95	$19.95	Foundations in Computer Studies With COMAL, John Kelly, *363 pages* *General* || Beginners text book, Jr/Sr High School level
-.-	$18.95	$20.95	Beginning COMAL, Borge Christensen, *333 pages* (imported from England) *General* || Beginners text book, Elementary School level, written by the founder of COMAL
-.-	$4.95	$6.95	COMAL From A to Z, Borge Christensen, *64 pages* *C64 0.14* || Mini reference guide to C64 COMAL 0.14 by the founder of COMAL
-.-	$12.95	$14.95	Captain COMAL Gets Organized, Len Lindsay, *102 pages with disk* *C64 0.14* || Application tutorial to illustrate benefits of modular programming
-.-	$4.95	$6.95	COMAL Workbook, Gordon Shigley, *69 pages* *C64 0.14* || Companion to the Tutorial Disk, great for beginners, full sized fill in the blank style
-.-	$12.95	$14.95	Graphics Primer, Mindy Skelton, *84 pages with disk* *C64 0.14* || Beginners graphics and sprites tutorial
Special:		$0.00	Free Doc Box with $20 or more of Doc Box style pages (expires 6/30/87, one per subscriber)
	$5.00		Doc Box only $5 with any one Doc Box pages book (offer expires 6/30/87)

---» Add the total for items on this side to the bottom of the other side. Shipping add $3 per book.

Then mail to: COMAL Users Group, USA, Limited, 6041 Monona Drive, Madison, WI 53716

ORDER FORM

Subscriber #_____ Name:_____

(more on other side)

April 1987 - Prices subject to change Street:_____

(only subscribers get lower prices)

Prepaid orders only, US Dollars, or charge: City/St/Zip:_____

VISA/MC #:_____exp date:_____Signature:_____

To order, circle price or check box as indicated:

NEWSLETTER & DISK SUBSCRIPTIONS (and single issues)

$18.95 (6 issues) **COMAL Today** newsletter subscription--> [___]renew [___]start at current issue

$26.95 (10 issues) (Canada add $1 per issue; Overseas add $5 per issue)

$1.00 each Backissues COMAL Today->circle issues wanted: 5 6 7 8 9 10 11 12 13 14 15 16 17

$35.95 (6 disks) **Today Disk** subscription--> [___]renew [___]start at current [___]start at #_____

$55.95 (10 disks)

$9.75 each Today Disk -> circle disks wanted: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

SYSTEMS:

Sub price	List price	Item Description - disks are Commodore format unless otherwise stated
$27.95	$29.95	C64 COMAL 0.14 Starters Kit (*4 disks, 2 books, 6 newsletters, more*)-(ship $4)
$128.90	$138.95	C64 COMAL 2.0 Cartridge Deluxe Pak (*cart, superchip, 2 books, 4 disks*) (ship $5)
$89.95	$99.95	C64 COMAL 2.0 Cartridge plain (*no manual, no disks*) (ship $2)
$24.95	$29.95	Super Chip for black C64 COMAL 2.0 Cartridge (*add $5 for installation/testing fee*) (ship $1)
$185.00	$199.95	**New-C128 COMAL 2.0 Cartridge** (*cartridge, manual, demo disk*) (ship $5) (special order)
$9.75	$14.95	Commodore PET 32K computer COMAL 0.14 (*disk only*)
$595.00	$595.00	UniComal IBM PC COMAL 2.1 (*English manual*) (special order only) (ship $5)
$4.00	$5.00	**New-Mytech IBM PC COMAL 2.0 Demo disk** (due June 1987) (near final is shipping NOW)
-.-	-.-	Coming soon...CP/M COMAL 2.0, Mytech IBM PC COMAL 2.0, Apple COMAL

DISK SETS:

$24.95	$29.95	**New-Full set of European 2.0 Program disks** (3 England & 9 Holland) (ship $2)
$14.95	$24.95	All four Cart Demo Disks (1,2,3,4) (ship $2)
$24.95	$29.95	**Full set of eleven 0.14 User Group disks** (1,2,3,4,5,6,7,8,9,10,12) (ship $2)
$14.95	$19.95	**Full set of four 2.0 User Group disks** (11,13,14,15) (ship $2)
$19.95	$24.95	Super Chip On Disk **plus Super Chip Programs Disk**
$49.95	$99.95	New-Source Code to Super Chip (requires Commodore's Assembler & Packages Library Fixes)

EUROPEAN 2.0 PROGRAM DISKS: ($9.75 each) (check box to order)

[___]England#1 [___]Holland#1 [___]Holland#4 [___]Holland#7

[___]England#2 [___]Holland#2 [___]Holland#5 [___]Holland#8

[___]England#3 [___]Holland#3 [___]Holland#6 [___]Holland#9

USER GROUP & CART DEMO DISKS ($9.75 each) (check box to order)

[___]UserGroup#1(0.14) [___]UserGroup#6(0.14) [___]UserGroup#11(2.0) [___]Cart Demo#1(2.0)

[___]UserGroup#2(0.14) [___]UserGroup#7(0.14) [___]UserGroup#12(0.14) [___]Cart Demo#2(2.0)

[___]UserGroup#3(0.14) [___]UserGroup#8(0.14) [___]UserGroup#13(2.0) [___]Cart Demo#3(2.0)

[___]UserGroup#4(0.14) [___]UserGroup#9(0.14) [___]UserGroup#14(2.0) [___]Cart Demo#4(2.0)

[___]UserGroup#5(0.14) [___]UserGroup#10(0.14) [___]**UserGroup#15(2.0)**

SPECIALTY DISKS ($9.75 each) (check box to order)

[___]**DataBases**(0.14/2.0) [___]TutorialDisk(0.14) [___]Utility #1(0.14) [___]**SuperChip Programs**

[___]Font(0.14/2.0) [___]AutoRunDemo(0.14) [___]Utility #2(0.14) [___]Shareware2.0(3 sides)

[___]Games(0.14/2.0) [___]ParadiseDisk(0.14) [___]SlideShow#1&2(0.14) [___]Read & Run(2.0)

[___]Modem(0.14/2.0) [___]BricksTutorials(0.14) [___]SpanishCOMAL(0.14) [___]Math & Science(2.0)

[___]**New-Full 0.14 Sampler** from San Francisco show [___]Articles#1(textfiles) [___]Typing(2.0)

Total $_____ + $_____ shipping (this side) -- **Shipping is $3 per book**

Total $_____ + $_____ shipping (other side) -- **Minimum shipping charge of $2 per order**

Total $_____ + $_____ = US$_____ **Total Paid** (WI add 5% sales tax) (overseas is extra)

Mail to: COMAL Users Group USA, 6041 Monona Drive, Madison, WI 53716 or call (608) 222-4432

Midnite Software Gazette

P.O. Box 1747
Champaign Illinois, 61821
Phone (217) 356-1885

Hello and welcome to the Midnite Software Gazette! We are the oldest independent magazine in North America for users of Commodore brand computers.

The Midnite Software Gazette was born to provide a forum for reviews of hardware and software for Commodore computers. Before becoming a commercial magazine, Midnite was published by Jim Strasma and myself as the newsletter for the Central Illinois Pet User's Group. Since that time, over seven years ago, we have published over 1600 reviews. And we are still publishing the same hard hitting reviews that the you need when looking for hardware and software.

The material in the Midnite Software Gazette is written by users--by the experts recognized throughout the Commodore world, and by the home hobbyist who has something to say. Literally hundreds of names have by-lined our reviews and articles, and we always invite you to contribute as well. We need your input as well as your support.

Sincerely,

Jim Oldfield, Jr., Editor-in-Chief.

MIDNITE SOFTWARE GAZETTE SUBSCRIPTION FORM

F. Name:_____ L. Name:_____

Street:_____ Apt. #:_____

City:_____ State:_____ ZIP:_____

One Year (12 Issue) Subscription: $ 23.00, $43.00 Air Mail. Payments are accepted in United States funds as check or money order. MasterCard and Visa are also accepted.
Pay by: Check:_____ Money Order:_____ MstrCrd:_____ VISA:_____

Credit Card Number:_____

Exp. Date:_____ Signature:_____

Back issues (12-15, 17-21, 23-27) available at half of cover price

---

MIDNITE SOFTWARE GAZETTE REVIEW FORM

PRODUCT:_____ Author:_____
Price:_____ Media:_____ Type/Application:_____
For computer:_____ Company:_____
Required Equip:_____ Optional Equip:_____
Protected?___ How?_____ Warranty?_____
Similar to:_____ Compatible with:_____

In 250 to 500 words, describe the program, tell what you liked, what you did not like, what standard features are/are not implemented, and who should buy it. Then, considering how well it works, its price, and compatibility, state whether the product is NOT RECOMMENDED, AVERAGE, RECOMMENDED, or HIGHLY RECOMMENDED. Include your name, address, and telephone number.

MicroPACE, Inc., will pay $10 per review published at the time of publication, or, upon request, credit $10 to your subscription. Be timely, be detailed, but be CONCISE!

Mail all subscriptions, requests, and reviews to:
MIDNITE SOFTWARE GAZETTE
P.O. BOX 1747
CHAMPAIGN, IL 61820

Reviews may be uploaded to Starship BBS at (217) 356-8056, or to Compuserve: 76703,4033; Q-Link: MIDNITE; Delphi: MIDNIT

DATACOLLISION- sprite/data collision detect
datacollision «sprite#»,«reset collision flag?»
datacollision 3,true

DEFINE - set up a sprite image for later use
define «shape#»,«64 byte string def»
define 14,shape$

HIDESPRITE - turn off specified sprite
hidesprite «sprite#»
hidesprite 2

IDENTIFY - assign a shape to a sprite
identify «sprite#»,«shape#»
identify 2,14
(note: sprite 7 is used for the turtle)

PRIORITY- does data have priority over sprite?
priority «sprite#»,«data priority?»
priority 2,false

SPRITEBACK - set 2 multicolor sprite colors
spriteback «color1»,«color2»
spriteback 2,6 //red & blue

SPRITECOLLISION- sprite/sprite collsn detect
spritecollision «sprite#»,«reset collisn flag?»
spritecollision 2,false

SPRITECOLOR - set color of sprite
spritecolor «sprite#»,«color number»
spritecolor 2,6 //sprite 2 red

SPRITEPOS - position sprite at x,y location
spritepos «sprite#»,«x coord»,«y coord»
spritepos 2,160,99// x=160 & y=99 position

SPRITESIZE - set sprite size (expand or not)
spritesize «sprite#»,«x expand?»,«y expand?»
spritesize 2,true,true //double size

Turtle Graphics

BACK - move turtle backwards
back «length»
back 50

BACKGROUND- set screen background color
background «color number»
background 2 // red

BORDER - set the screen border color
border «color number»
border 0 // black

CLEAR - clear the graphics screen
clear
clear

DRAWTO - draw a line from current point
drawto «x coord»,«y coord»
drawto 50,80

FILL - fills in area with current color
fill «x coord»,«y coord»
fill 50,80

FORWARD - move turtle foward
forward «length»
forward 100

FRAME - set up a screen window
frame «left»,«right»,«bottom»,«top»
frame 0,319,0,199 // the default

FULLSCREEN - fullscreen graphics (f5)
fullscreen
fullscreen

GETCOLOR - returns color of specified pixel
getcolor(«x coord»,«y coord»)
print getcolor(50,80)

HIDETURTLE - make turtle invisible
hideturtle
hideturtle

HOME - put the turtle in its home position
home // x=160 & y=99 is home
home

LEFT - turn turtle left
left «degrees»
left 90 // a right angle

MOVETO - move to loc without line
moveto «x coord»,«y coord»
moveto 50,80

PENCOLOR - set turtle drawing color
pencolor «color number»
pencolor 2 // red

PENDOWN - put pen down, turtle draws
pendown
pendown

PENUP - pick pen up, turtle does not draw
penup
penup

PLOT - plot a point in current color
plot «x coord»,«y coord»
plot 50,80

PLOTTEXT - put text on graphics screen
plottext «x coord»,«y coord»,«text$»
plottext 0,24,"press space to continue"

RIGHT - turn turtle right
right «degrees»
right 180 // reverse direction

SETGRAPHIC - turn on graphics screen
setgraphic [«type»]
setgraphic 0 // hi-res screen
setgraphic //use previous graphic screen

SETHEADING - set turtle heading
setheading «degrees»
setheading 180

SETTEXT - turn on text screen (f1)
settext
settext

SETXY - set turtle x, y coordinates
setxy «x coord»,«y coord»
setxy 50,80

SHOWTURTLE - make turtle visible
showturtle
showturtle

SPLITSCREEN- 2 text lines above graphics
splitscreen // or use (f3)
splitscreen

TURTLESIZE - set turtle size (0 to 10)
turtlesize «size»
turtlesize 6